AD-A201 042

(4)

UNCLASSIFIED DTIC FILE COPY

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER AIM 1070 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) An Operating Environment for the Jellybean Machine | | 5. TYPE OF REPORT & PERIOD COVERED memorandum |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) Brian K. Totty | | 8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0622 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209 | | 12. REPORT DATE May 1988 |
| | | 13. NUMBER OF PAGES 156 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217 | | 15. SECURITY CLASS. (of this report) UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

DTIC ELECTE DEC 0 1 1988 S D E

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

| | |
|---|---|
| Operating systems, | distributed systems, |
| jellybean machine, | networks, |
| parallel processing, | virtual memory, multiprocessors, [illegible] |
| ensemble machines, | computer programs |

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see back of page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601 | UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 cont'd

The Jellybean Machine is a scalable MIMD concurrent processor consisting of special-purpose RISC processors loosely coupled into a low latency network. The problem with such a machine is to find a way to efficiently coordinate the collective power of the distributed processing elements. A foundation of efficient, powerful services is required to support this system.

To provide this supportive operating environment, I developed an operating system kernel that serves many of the initial needs of our machine. This Jellybean Operating System Software provides an object-based storage model, where typed contiguous blocks act as the basic metric of storage. This memory model is complemented by a global virtual naming scheme that can reference objects residing on any node of the network. Migration mechanisms allow object relocation among different nodes, and permit local caching of code. A low cost process control system based on fast-allocated contexts allows parallelism at a significantly fine grain (on the order of 30 instructions per task).

The system services are developed in detail, and may be of interest to other designers of fine grain, distributed memory processing networks. The initial performance estimates are satisfactory. Optimizations will require more insight into how the machine will perform under real-world conditions.

[illegible]

Massachusetts Institute Of Technology
Artificial Intelligence Laboratory

A.I.Memo No. 1070 May, 1988

# AN OPERATING ENVIRONMENT FOR THE JELLYBEAN MACHINE

Brian K. Totty

## ABSTRACT

The Jellybean Machine is a scalable MIMD concurrent processor consisting of special-purpose RISC processors loosely coupled into a low latency network. The problem with such a machine is to find a way to efficiently coordinate the collective power of the distributed processing elements. A foundation of efficient, powerful services is required to support this system.

To provide this supportive operating environment, I developed an operating system kernel that serves many of the initial needs of our machine. This Jellybean Operating System Software provides an object-based storage model, where typed contiguous blocks act as the basic metric of storage. This memory model is complemented by a global virtual naming scheme that can reference objects residing on any node of the network. Migration mechanisms allow object relocation among different nodes, and permit local caching of code. A low cost process control system based on fast-allocated contexts allows parallelism at a significantly fine grain (on the order of 30 instructions per task).

The system services are developed in detail, and may be of interest to other designers of fine grain, distributed memory processing networks. The initial performance estimates are satisfactory. Optimizations will require more insight into how the machine will perform under real-world conditions.

Submitted to the Department of Electrical Engineering and Computer Science on May 6, 1988 in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science.

Thesis Supervisor: William J. Dally
Title: Assistant Professor of Electrical Engineering and Computer Science

Keywords: Operating Systems, Jellybean Machine, Parallel Processing, Distributed Systems, Networks, Virtual Memory, Ensemble Machines

**This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. The research described in this paper was supported in part by the Defense Advance Research Projects Agency of the Department of Defense under contracts N00014-80-C-0622 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.**

## Acknowledgements

Now that I am polishing up my thesis, I want to take the time out to thank a few of the people who helped me complete this task.

Thanks to my thesis advisor, Bill Dally, for providing a wealth of knowledge and the obligatory prodding that was necessary for timely completion of my thesis. Thanks for spending the time to repeat ideas hundreds of thousands of times until I caught on. Thanks for the help, advice and support. Mainly, thanks for the opportunity of working in your group. It was by far the most rewarding academic experience I had at M.I.T. My newfound enthusiasm will have significant effects on my future plans. Good luck with the project.

Thanks to all the members of the Concurrent VLSI Architecture Group for sharing your insight and time with me. Thanks, Andrew Chien, for your leadership, interest and advice, and your jellybeans. Thanks to Stuart Fiske for taking the time to think through things with me and for helping me back onto the track when I get confused. Thanks for your advice, your sense of humor, and for being an all around great guy. However, I won't mind too much if you quit the singing. And "gobs" of thanks to Soha for all the support and advice and assistance and most of all for being a great friend. Thanks for the ideas, suggestions, and for inciting heated political discussions. Unfortunately, you call my car a "toy", so I won't acknowledge you any more. Thanks to Waldemar Horwat for his ideas and advice. Thanks for providing the only readable documentation when I entered the group and for the simulator which my system was tested on. Thanks to Jerry Larivee for providing the technical support, for his thoughts on garbage collection, and for being a funny guy. Thanks to Paul Song for sharing his knowledge of networks, for participating in four hour discussions on twentieth century Middle Eastern history, and for being a "yogurthead". Go home to your family, will you! And thanks to Scott Wills, for providing the intellectual diversity and organization to the group. Thanks for your interest and your advice. Finally, thanks to Anant Agarwal for sharing his expertise, his humor, his knowledge of Indian history, and his advice. And thanks, of course, for splot.

Scott Heeschen deserves thanks, for suggesting I approach Bill Dally for a UROP. Let me thank my history professor Hasan Kayali in advance for letting me turn my history paper in late in order to complete my thesis. Thanks also for providing my most interesting class this term. Thanks to all of the members of the LAND-OF-THE-BIZARRE@OZ mailing list for the consistant stream of mentally deranged mail, that helped to keep my spirits high, as well as the size of my mail files.

Thank you to my relatives, who were so proud to see me go to M.I.T., and who made my holiday gift buying much easier with their insatiable desire for M.I.T. insignia. And finally, thank you to my parents, who put up with the financial and emotion burden of sending me to college. Thank you for instilling in me the respect of education and the desire for understanding. Thank you for your care and support and advice, even if I sometimes don't seem to appreciate.

To my friends and family, best wishes, and take care.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

*I am the people — the mob — the crowd — the mass*
*Do you know that all the great work of the world is done through me?*

— Carl Sandburg, in *I Am the People, the Mob* (1916)

*Power is the great aphrodisiac.*

— in *The New York Times* (January 19, 1971)

Concurrent processing is becoming a progressively more popular field in computer science. The vision of harnessing previously undreamt of computational power at a reasonable cost is leading the drive. By connecting many moderately powerful microprocesors in a communications medium, system designers hope to be able to take advantage of the collective power of the architecture to solve tasks that were previously time or cost-prohibitive.

Unfortunately, the eager concurrent system designer soon finds that many issues are still unresolved. Though people have a fairly good grasp of ways to build successful sequential machines, it is less clear how to build optimal, or even acceptable concurrent systems. The designer is soon faced by a barrage of questions that are difficult to answer. "What grain of parallelism should be supported?" "What level of functionality should the processors provide?" "How should the processors communicate?" "How tightly coupled should the processors be?" "How should memory be managed?" "How should the load be distributed?". Many research groups are attempting to answer these questions at this very moment.

Some insight into concurrent architectures has been gained over the years, and the current directions of research reflects the knowledge gained. Multicomputer networks (sometimes called "ensemble machines") are one direction that concurrent systems research has taken. This genre of machine connects relatively conventional microprocessors via an automatically routed network. The design is advantageous because it takes advantage of well understood sequential processor technology for the processing nodes, and the performance of the system can grow proportionately with the number of processors[1], providing *scalability*.

For the past two years, the Concurrent VLSI Architecture Group at M.I.T. has been designing a concurrent processing network, christened the Jellybean Machine, under the direction of Professor William Dally [Dal86c]. The goal of the Jellybean Machine project is to design a scalable concurrent processor out of low-priced (jellybean) parts, that efficiently supports an object-oriented execution model. The processor is targeted at both symbolic and numeric applications, and will be programmed in high-level, object-oriented languages. It hopefully will serve as a succesful example and a test bed for advanced concurrent systems research.

## 1.1 Scope of Thesis

This thesis report describes the design and implementation of an operating system prototype for the J-Machine. The operating system was required to support a global namespace across the distributed processors, allocate memory in an object-based storage model, support

[1]at least up to some point.

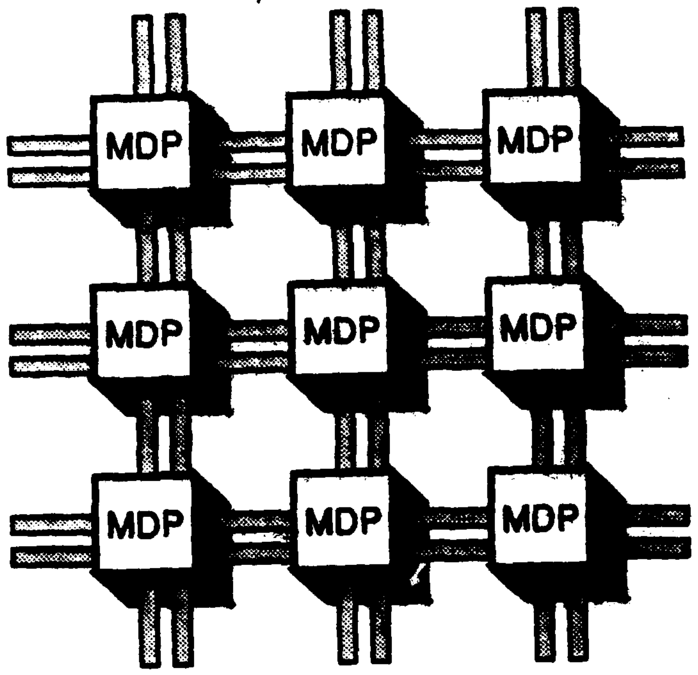
inter-processor communication, provide system services to control code execution, object migration, and an object-oriented calling model. It also provided a perch from which more advanced issues in system design could be studied.

## 1.2 Highlights of Contributions

In the course of the design of the J-Machine operating system, several ideas were developed that may be of special interest to the designer of multicomputer networks.

- In section 3.4, I describe a virtual addressing system that resolves objects names across distributed nodes by a mechanism known as *hometown addressing*. This scheme delegates to object birthnodes the responsibility for knowing current object residences, permitting object migration. An accompanying mechanism of "hints" is provided to improve performance.
- To simplify the hardware with minimal cost in flexibility, we have developed an explicit, one time virtual translation scheme via the XLATE machine instruction, that converts a virtual address to a physical one. Retranslation is provided for automatically by fault handlers.
- Chapter 5 describes a low overhead code execution model that supports inexpensive remote procedure calls, local caching of code, and convenient suspension and resumption of processes.
- Section 5.4 describes a system for fast context creation that involves the re-use of old context objects. This is an important optimization based on the short life and rapid freqency of context allocation.
- Section 5.6 outlines a simple and fast, resource distribution mechanism that limits bottlenecks and cross network traffic by dynamically creating a type distribution tree for the resource.

## 1.3 A Closer Look At The Jellybean Machine



The J-Machine is composed of many custom RISC microprocessors called Message-Driven Processors or MDPs. These processing elements have small, local memories and are connected in a loosely coupled network. Inter-node communication is provided via message sends that are automatically routed to the proper destination nodes. A virtual object-based memory abstraction is built over the distributed nodes providing a uniform global namespace. Various levels of low-cost execution control provide a reasonably fine grain of concurrency (on the level of 30 instruction procedures). An object-oriented execution model is built upon this fine-grain execution model. The rest of the system implements miscellaneous system services and mechanisms to improve performance.

## 1.4 Background

Concurrent architecture design has been seriously studied for at least the past fifteen years, but there is still much to be learned. The various visions of machines, operating systems, and target applications are so diverse, that few definitive statements can be made.

We see SIMD parallelism, promoted by vector operations as seen in the Cray. More complicated architectures like the Connection Machine [Hil85], and systolic array processors like the Warp [Kun82] are alternative approaches, providing fine-grain concurrency with repetitive processing while permitting reconfiguration. MIMD architectures are just as diverse. There are extremely fine-grain dataflow machines like the Manchester Machine, Sigma-1, and the MIT Tagged-Token dataflow Machine [Aea80], bus-based shared memory architectures like the IBM RP3, Inmos Transputer, and C.mmp [WLH81], multicomputer networks like the Cosmic Cube [Sei85] and Cm* [OSS80] and distributed systems like System R* [Lin80].

The Jellybean Machine, while borrowing ideas from successful research endeavors, has goals unique enough to gain a somewhat different character from other machines of its genre. It communicates via message passing and addresses only local memory, as in the Cosmic Cube [Sei85] and the Medusa system [OSS80]. On the other hand, these two systems control execution by a system of pipes and locks, where processes wait for data to arrive via messages. The J-Machine, instead, uses message sends to schedule processes, and not to provide socket-to-socket communication. State manipulation doesn't involve explicit connections between running processes. Instead, return values are propagated around to slots in contexts and code is executed when results arrive in a more "functional" manner.

Many systems also have virtual memory and some systems use an object or segment based storage model [WLH81] as does the J-Machine, but the emphasis is slightly different in our design. Where most systems use a virtually addressed, multi-level memory system to expand primary memory and provide relative address mapping, the J-Machine uses a virtual addressing system to provide a global namespace across all nodes and to provide convenient access to objects as the primitive memory metric. This is more similar to large, complex distributed systems such as IBM's distributed database, System R* [Lin80] than conventional parallel processors.

Finally, the J-Machine targets itself to a high-level programming environment. The RISC processing node, called the Message-Driven Processor [HT88], provides a fast, powerful substrate for the execution of high-level languages, such as Smalltalk. There are several architectures designed for the efficient execution of high-level language applications, such as the Symbolics Lisp Machine and the SOAR Smalltalk processor [Ung87], but very little work has been done targeting *concurrent* processors to high-level languages.

## 1.5 Organization

The rest of this report will discuss the structure of the Jellybean system. Chapter 2 provides a high level layering of the Jellybean system — from single processing node hardware to the high level programming of the entire concurrent processing network. Chapter 3 describes the memory management and addressing system. Chapter 4 discusses the machine as a distributed system supporting object migration to balance load. Chapter 5 explains code execution on the method level, and 6 details the object-oriented calling extensions. Storage reclamation issues will be introduced in chapter 7. Chapter 8 discusses some of the services provided to support high-level language constructs and to control code execution. Chapter 9 describes the prototype operating system implementation noting its successful as well as not-so-successful features, and discussing some of the difficulties and quirks faced by the system designer. The report concludes with a performance evaluation and summary in chapters 10 and 11.

# Chapter 2

# The Execution Model of the Jellybean Machine

*These unhappy times call for the building of plans ...*
*that build from the bottom up and not from the top down*

— Franklin Delano Roosevelt, in his April 17, 1932 Radio Address

The Jellybean Operating System Software (JOSS) is built in a layered manner where each layer provides a different model of functionality to the machine. Figure 2.1 attempts to describe this layering, and what new functionality each layer provides to the entire system.

At the bottom of the figure lies the base processor and boot code. At this stage, the processing node can be initialized, and can run independently as a limited microprocessor. The addition of system call and fault handlers provide a level of system services and robustness to the microprocessor, allowing it to allocate memory in an object-based, virtually addressed manner, and to handle various types of exceptional conditions at run time. These first two levels of the Jellybean system build up the abstract processing node

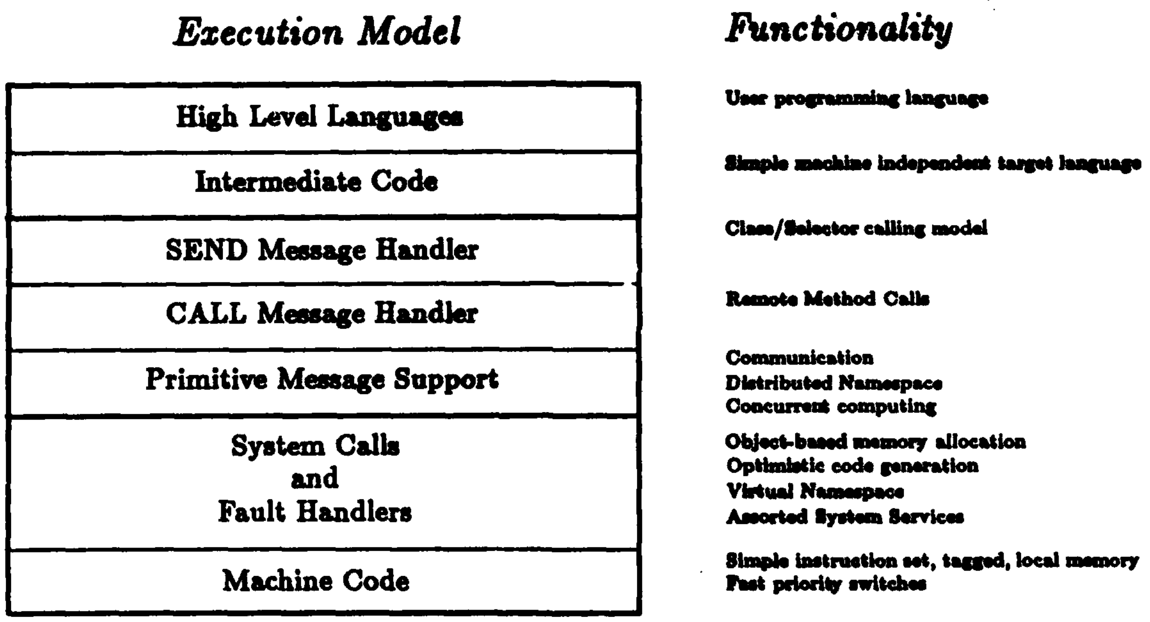


Figure 2.1: Layering of Jellybean System

capable of executing machine code and performing a set of system services.

Concurrency is provided as the next level of functionality by the introduction of *primitive message handlers*. Each processing node has the ability to send messages to any other node, where a *message* is simply a physical address to start running on a foreign node, followed by routine-specific data. Thus, a Jellybean *primitive message* is actually just a way of changing a program counter of a remote node. A set of common operations can be placed in identical physical memory locations on each node, so that an operation can be run on any node by mailing that routine's address to the node. The operating system provides a small set of *primitive message handlers* to perform common operations which reside in the same locations on each node. With this small set of locked-down routines, the machine gains the ability to compute concurrently, to use a global addressing abstraction over the physically distributed memories, and to perform some amount of object migration and other control of resources.

Two special primitive message handlers are special, in that other system services are built on top of them. The CALL message handler provides a mechanism for starting code contained in virtually-addressed relocatable objects, rather than just code that resides at locked-down physical addresses. This provides a convenient way of packaging objects and supporting remote procedure calls. The SEND message takes the code execution mechanism to an even higher level, and provides for a dispatch-on-type calling model as used in object-oriented systems like Flavors or Smalltalk.

The final two layers of the system are the interfaces for the programming models. The Jellybean Machine under this highest level of abstraction appears to the user a system to run high-level languages like Smalltalk.

The rest of this chapter will go into the abstractions in more detail, describing what functionality each level of the machine provides. It may be helpful to refer back to figure 2.1 as you read the following sections.

## 2.1 The Processing Node

Each node of the Jellybean multiprocessor (a *Message-Driven Processor*) is a tagged-architecture microprocessor with a small on-chip memory with separate register sets for operating at two priority levels.

### 2.1.1 Machine Code

The machine code interpreted by a Message-Driven Processor (MDP) is a simple 3 operand instruction set [HT88]. Code is executed sequentially, and changes in control are provided by simple conditional and unconditional branches. The instruction stream is accessed via two registers, one that points at the base of the code block (A0), and one that indicates the current offset into this block (IP).

### 2.1.2 System Calls

The processor also has a small fixed length stack, and a mechanism to make system calls. This provides us with the ability to change control to common subroutines, and easily restore execution upon return. The addition of the system call machinery gives us the ability to provide several extensions to the processor in terms of system services written in machine code. Heap management, and an object-based memory allocation model are provided with system calls, as are the mechanisms to address these objects with relocatable, virtual IDs.

### 2.1.3 Fault Handlers

Similar to system calls, the MDP also contains a fault handler table providing software routines to run when instructions fault because of various exception conditions (tag mismatches, addressing past segment, integer overflow, translation buffer lookup miss, etc.). When a fault occurs, the IP is pushed onto the stack, and the appropriate fault routine (found in the exception vectors table) is run. An address of each fault handlers is placed in the exception vector table by software initialization. The addition of the fault handlers gives us several advantages in our quest of an object-oriented concurrent processor. We can use tag checking to support optimistic code generation and a type of "generic operation" approach on the machine code level. The fault handlers also provide us the ability to efficiently implement virtual ID lookup via the XLATE instruction. The fault handlers will be described in more detail later when the entire system has been more thoroughly explained.

Since both the system calls and fault handlers are supported by a software initialized vector table, the processor can be "reshaped" into a different type of machine by replacing the ROM code that sets up this table. Only the instruction set is fixed, allowing the MDP processing node to be used as a basis for various alternative concurrent processing system paradigms.

### 2.1.4 The Basic Node of Computation

With what we have described so far, our processor is a sequential machine, able to be executing in one of two priorities. It refers to its instruction stream using physical memory base and offset registers. The addition of the system calls provides an interface to OS services, such as those to allocate memory, generate virtual object IDs and to manage object ID to physical address translation. The fault handlers permit us to develop "optimistic" code, where a normal, error-free execution will proceed rapidly, and we only pay the price of software execution if an error condition occurs. The fault handlers are also used to support a fast virtual namespace, where translation can be as fast as the XLATE instruction.

The sum is a flexible, object-based microprocessor that will serve as our basic node of computation as we venture into the realm of concurrency.

## 2.2 The Concurrent Processor Model

By providing mechanisms for node-to-node communication, our machine becomes a multiprocessor, called the Jellybean Machine. Many MDP processing nodes (as well as other potential nodes such as floating point processors and memory nodes) are connected together in a network. Communication between the nodes is provided by the MDP SEND instruction which injects messages into the network. The messages are routed by routing hardware to the message queues on the destination node.

Messages received by an MDP processing node consists of two parts, a message header which contains the address of the primitive message handler to run, and a sequence of message specific data words. The header of the message acts in effect like a process descriptor for providing efficient message execution. When a message arrives at the specified node, it lands in the destination node's queue. The queue acts as a FIFO scheduler of primitive message processes. When the message moves to the head of the queue, the MDP executes the message by setting the instruction pointer register to point to the primitive message handler whose address is in the header of the message.

Several useful system services are written as primitive message handlers. Examples of primitive message handlers include those to make a new object on a node (NEW_MSG) and to request a copy of a method from a node (METHOD_REQUEST_MSG).

With the addition of primitive messages, we have the ability to process concurrently, and to support a distributed namespace. We can now extend our virtual memory system to support naming of objects, not just in the local memory, but on any node in the entire network. With a distributed namespace, we gain flexibility of resources. We can migrate objects as we need them to balance load and to free up memory.

### 2.2.1 Methods and the CALL Message

Up to this point, we have only been able to run foreign code that resides at fixed physical locations. We desire a more flexible mechanism for dealing with blocks of code, such as those that will be output by compilers. Since we already have an object based storage model, it would be very convenient to store code routines in objects and provide a mechanism for their execution. We call code routines stored in virtually addressed, relocatable objects *methods* to differentiate them from physical locked down code sequences. We provide a mechanism to start these methods executing by writing a primitive message handler called the CALL message handler. When a CALL_MSG starts executing on a node, it runs the method indicated in the message argument. This allows us to have a flexible system of remote procedure calls.

### 2.2.2 SENDing Selectors to Objects

The final operating system layer in our quest for an object-oriented execution model is the SEND_MSG message handler. A SEND_MSG consists of a selected generic operation, represented by a unique symbol called a *selector*, followed by the object(s) that the selector acts upon. If we wanted to send the *DRAW* selector to an object (say a triangle), we would SEND a SEND_MSG message to the node the triangle object resides on, passing the selector *DRAW*, and the virtual address of the triangle object receiving the selector (called the *receiver*). When the SEND_MSG handler gets executed, it determines the appropriate method to run, and then remotely calls the procedure by sending a CALL_MSG message to this method which then draws the triangle.

In order for this system to work it is necessary to maintain certain system tables that map pairs of selectors and object classes with the virtual IDs of methods to perform the desired information. It is also necessary to insure that semantically indentical selector operations get the same selector symbol. In other words, all PLUS operations must get the same symbol representing +. The exact mechanisms of the class/selector system will be described in more detail in chapter 6.

## 2.3 High Level Language Model

For the final part of our tour of the Jellybean Machine, let us step back once more, and view the machine from the perspective of the programming languages that will be used to write user programs.

### 2.3.1 Intermediate Code

To provide a uniform target language for compilers, we have specified an intermediate language called *i-code*. This language has a simple set of operations, and a simple manner of referencing operands. By passing the send code through a code generator and a linker/loader we can store actual MDP machine code on nodes. The i-code level of the system provides a convenient entry point for various compilers that necessitates no knowledge of the underlying layers. All interaction is via the protected subsystem of the i-code interface. This interface, in effect, provides an abstract *i-code machine* that can be of use in many different machine configurations. Implementations of this interface on different machine architectu.es would provide a convenient way to reuse compilation tools and compare system performance.

### 2.3.2 User Languages

The user language model is what would be seen by the user of the Jellybean Machine. He/she would be faced with the language interaction shell and would see none of the internal layers that compose the system. The currently supported user language is a prefix notation form of concurrent Smalltalk [DC]. Other languages, such as a Lisp with flavors should also be possible.

# Chapter 3

# Memory Management and Addressing System

*Work without hope draws nectar in a sieve*
*And hope without an object cannot live*

— SAMUEL TAYLOR COLERIDGE, in *Work Without Hope*

*Oh call it by some better name*
*For friendship sounds too cold.*

— THOMAS MOORE in *Ballads and Songs: Oh Call It by Some Better Name*

The Jellybean Machine, targeted for object-oriented applications, needs to have an object-based storage model. This chapter sketches the machinery that interact to provide this model. The mechanisms basically consist of two parts, (1) the services to allocate and deallocate contiguous blocks of physical memory, and (2) the virtual addressing abstractions that make objects the basic unit of storage. This virtual address allows object relocation and provides a way to reference storage on foreign nodes. Virtual naming and physical allocation systems combine to form an object based programming system.

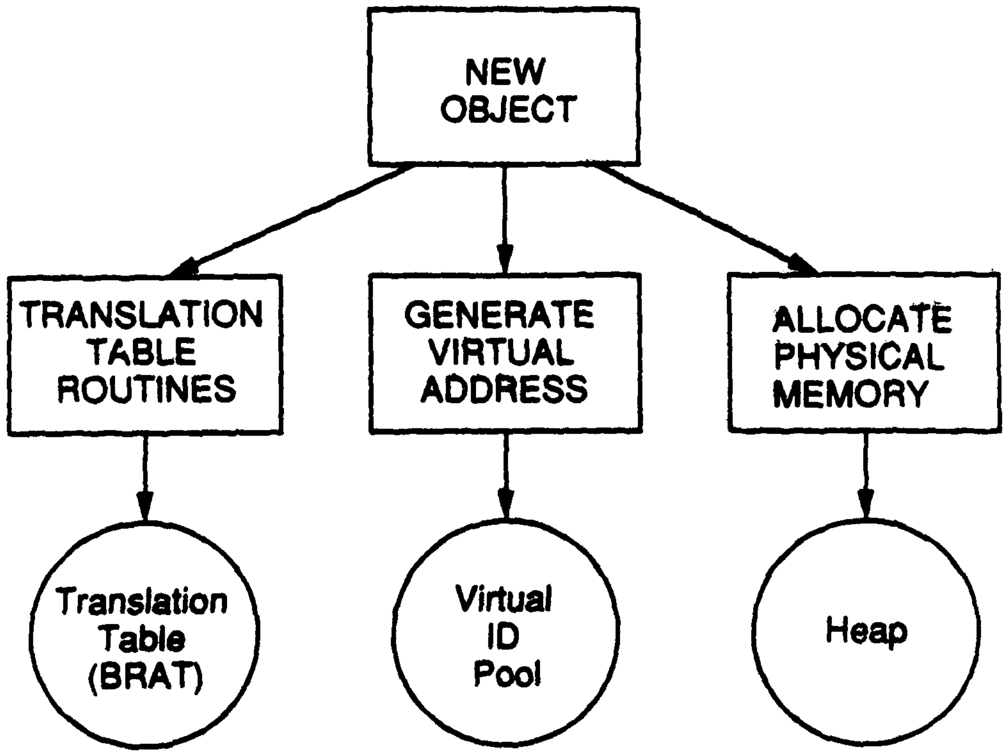


Figure 3.1: Schematic Model of the Memory System

At the heart of the object based system is the NEW system call, which creates a new object. This routine utilizes the 3 object system subsystems, the translation manager, the name manager, and the memory manager. This interaction of the various systems is shown in figure 3.1.

## 3.1 "Freetop" Contiguous Heap Allocation

Each node of a Jellybean Machine has its own local memory that can be accessed very rapidly. Part of this local memory is reserved as a heap to allocate blocks of memory from. Heap allocation is done in a straightforward "freetop-next" manner. Memory is allocated starting from the current top of free memory, and the freetop pointer is moved past the block allocated. The ALLOC system call handles the allocation requests.

## 3.2 Compaction is Fast

Deletion of objects fragments the heap leaving unused "holes" in the heap. We reclaim this storage by sweeping objects down toward the base of the heap, to fill up the blank space, with the freetop following accordingly. Since each local memory is small and fast, and each processor can sweep in parallel, compaction takes very little time. Figure 3.2 shows a process of heap allocation, deletion, and compaction.

## 3.3 Physical Base/Length Addressing

Blocks of memory are described by physical base/length values supported by the processor's primitive ADDR data type. The base is the starting address of the block of memory, and the length is used for access bounds checking. The format of an ADDR tagged value is shown in figure 3.3. The tag of the physical address word is a unique number ADDR representing a physical address value. The R bit is used to specify that an address value points to a relocatable object. The I bit specifies that the address is now invalid. Both of these bits are used for the implementation of virtual addressing.

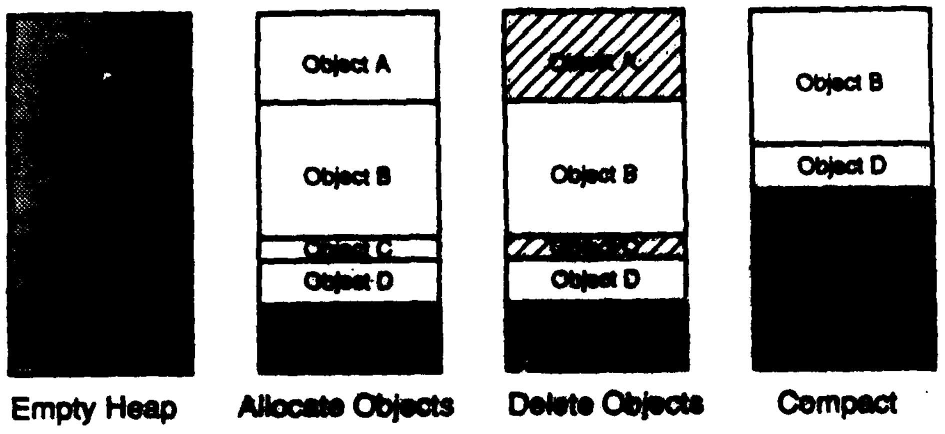


Figure 3.2: "Freetop" Heap Allocation, Deletion, Compaction

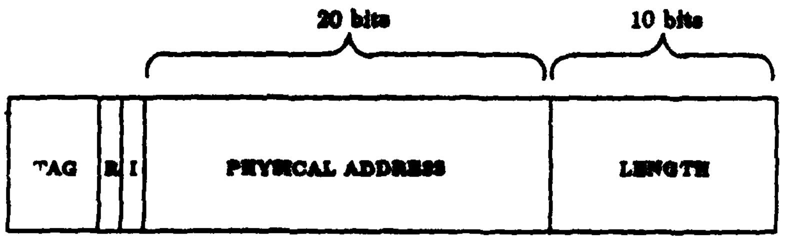


Figure 3.3: A Physical Address Word Format

## 3.4 Virtual Addressing Extension

Having physical addresses only allows us to access objects on the current node. It provides us no mechanism for naming objects on different nodes. For this reason and because it eases relocation and provides an object-based storage model, we extend our addressing system from the local, physical namespace provided by the physical ADDR values to a global, virtual namespace using virtual object IDs. A virtual ID is a global name for an object.

### 3.4.1 Creating New Objects

Objects are created by the NEW system call. The system call allocates memory with the ALLOC call, reserving the first two words of the allocated block of memory for object header information. Once the block of memory is allocated, a unique, virtual ID is generated with the GENID system call. The first word of the block of memory is initialized to contain the length and data type of the object, and the second word is set to the virtual ID. Finally, a virtual ID to physical address binding is made for the object so we can find the physical location given the ID. The format of an object is shown in figure 3.4.

To manage this virtual namespace efficiently, we need some operating system and hardware support. First of all, the processor provides a matching ID register for each physical address (A) register. These ID registers hold the virtual IDs for the objects whose physical addresses are in the A registers. We also provide a translation buffer as we will discuss shortly.

### 3.4.2 Virtual Memory System Calls

The GENID system call generates a new serial number, unique on the current node. The current implementation encodes a virtual ID in two fields, a node-unique serial number, and a node number component representing the node number an object was created on. The

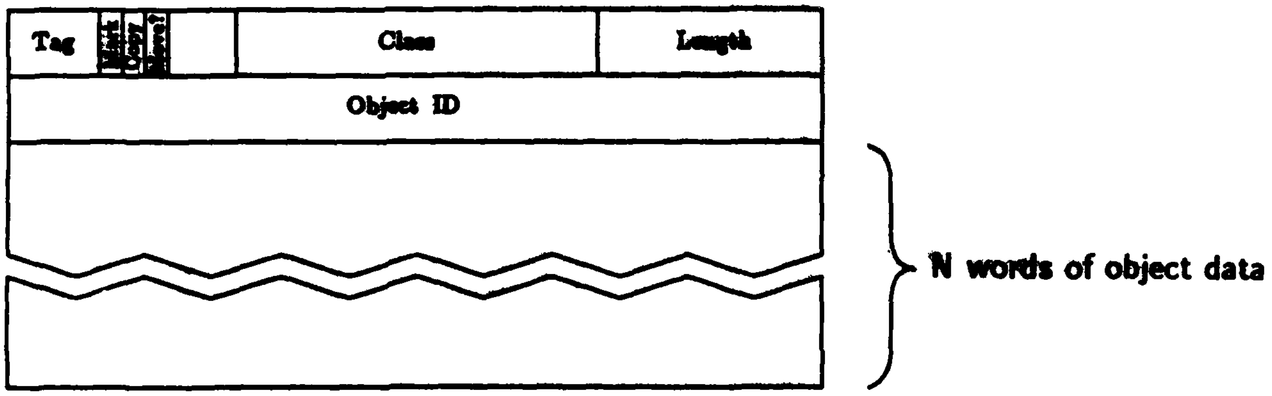


Figure 3.4: The Structure of an Object

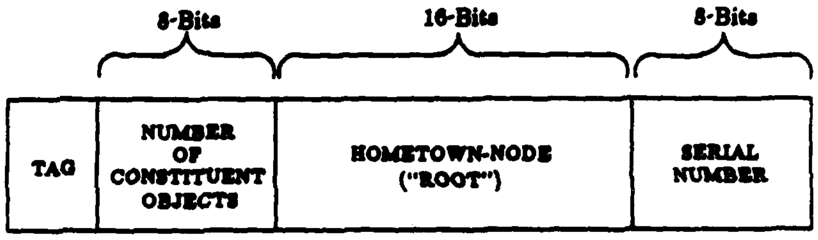


Figure 3.5: A Virtual Address Word (ID) Format

format of this virtual ID is shown in figure 3.5. There are also several utility routines used to manage the virtual → physical translation table (called the Birth/Residence Address Table, or BRAT). These routines add, lookup, and remove bindings from the translation table. They are implemented by the extended system calls BRAT_ENTER, BRAT_XLATE, and BRAT_PURGE respectively. Finally, we provide the NEW system call to allocate and install a new object. This service allocates physical memory, generates a virtual ID, installs the virtual → physical binding in the BRAT, and returns both the ID and the address. The NEW system call is to the virtual addressing model as ALLOC is to the physical addressing model.

### 3.4.3 Translation Buffer

To speed up translation, each processing node has a 2-way set-associative translation buffer, and the accompanying ENTER, XLATE, and PURGE machine instructions. The XLATE instruction will fault if no binding is found in the cache, and a software exception handler will be run to resolve the name.

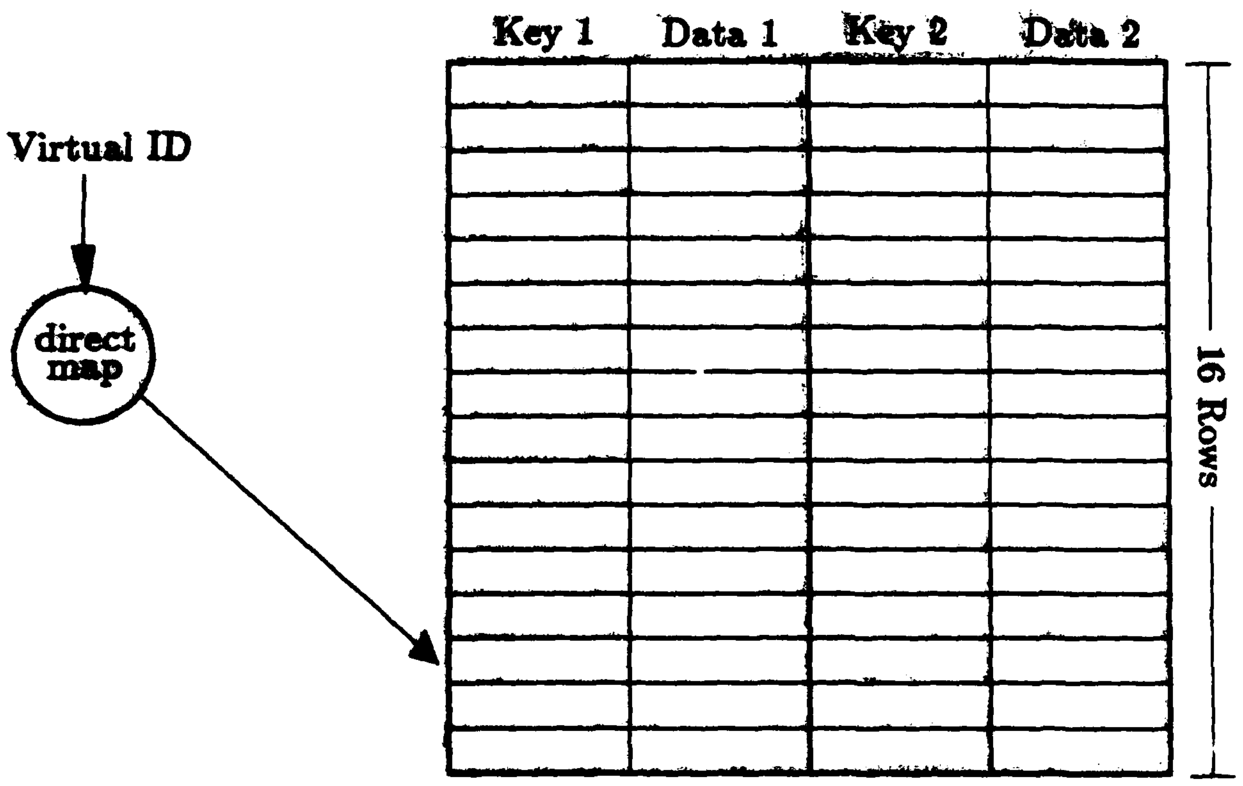


Figure 3.6: Format of the Translation Buffer

### 3.4.4 Automatic Retranslation

To support maximum efficiency in normal case situations, the processing node provides an "invalid" bit in each address (A) register. If this bit is set, it signifies that the ID and A register have values that are no longer consistant. Any access of an invalid A register will cause a fault handler to be run which will retranslate the ID register into the A register and continue. This way we can be "lazy" and retranslate invalid bindings only if needed.

## 3.5 Summary

Physical block allocation is used to reserve segments of memory. Virtual IDs are associated with these blocks of memory, and bindings are formed, to provide an "object-based" allocation model. This object allocation model provides the following benefits

- An abstract memory model, where "objects" are the primitive metric of storgae rather than physical addresses.
- A location independent memory model with indirection through a translation table, allowing ease of relocation.
- The ability to represent the data types of objects.
- The introduction of a *global namespace* where we can refer to objects residing on any node of the network.

# Chapter 4

# Distributed System Support

*I pity the man who can travel from Dan to Beersheba and cry, 'Tis all barren!*

— LAWRENCE STERNE, in *A Sentimental Journey* (1768)

In the previous chapter we developed a object based allocation model and a global naming system. With this functionality, we gain much greater flexibility. We take this system one step further in this chapter, as we describe a mechanism to migrate objects from node to node. This added ability requires a few extensions to the virtual naming model presented in the previous chapter.

## 4.1 The Idea

In the previous naming model, virtual IDs were bound to physical addresses. Since objects were not allowed to migrate, they were forced to always reside on their birthnode. Now that objects are allowed to emigrate to different nodes, we need to expand our name resolution system. In addition to virtual → physical bindings we add a virtual → node-number binding semantically representing a "hint" that the object in question now resides on a

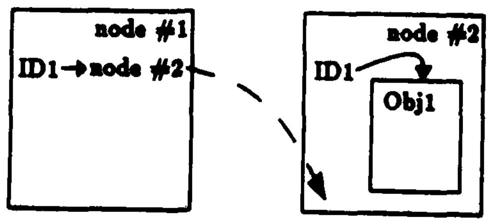


Figure 4.1: An Example of Hints

different node number. Figure 4.1 shows that node #1 has a hint that an object is on node #2.

## 4.2 Chaining of Hints

These node number "hints" indicate another node to look on for the object in question. The current implementation allows chaining of hints (although cycles will never form). If we ever follow a path of hints and find no binding for the object ID, we then query the birthnode which is required to have a path to the object in question. Figure 4.2 is a snapshot of a system where a chain of hints has formed to an object.

A question then arises as to how long to let these chains of hints be. Some distributed systems, such as System R* [Lin80], only allow paths of length 1, i.e. one hint. If the

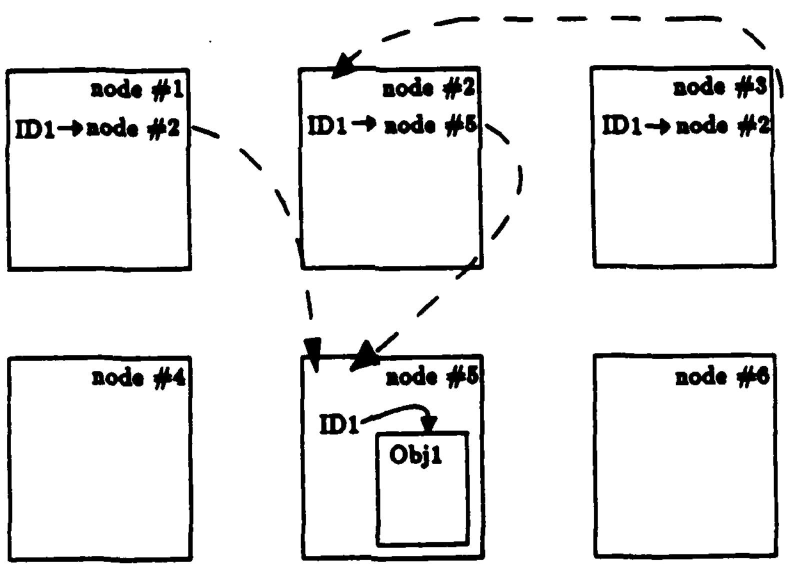


Figure 4.2: Chains of Hints

object is not one hint transition away, the system then defaults to the birthnode where the location of the object is found, and the previous incorrect hint is updated. However, in our system we choose to have multiple hints because objects may migrate quite a bit, and this would increase the number of birthnode accesses. Performance could significantly degrade if a popular object moved quite a bit (as we would expect popular objects to do). If we notice in later performance experiements, that chains of hints become commonplace, adding latency and unnecessary network traffic, we can adopt one of 2 solutions, (1) only allow one hint or (2) collect and update old hints periodically.

## 4.3 Calculating Likely Nodes From Object IDs

The operating system provides a system call for finding a likely node that an object resides on. This ID_TO_NODE call takes the virtual ID of the object and returns a node number. It does so by the algorithm charted in figure 4.3. It works in the following way. The virtual ID is looked up in the translation table. If it is not there, we have no idea where the object is, so we check the birthnode. If there is a binding, but the binding is to a hint (an integer value), we return this hint as the probable residence node. Finally, if the binding is to a physical address, the object is local, and the local node number is returned.

## 4.4 Virtual To Physical Translations In The Migrant Object World

Now that objects are allowed to wander aimlessly across the nodes of the Jellybean Machine, virtual to physical address translations are necessarily slightly more sophisticated. Three conditions can occur when we attempt to translate a virtual ID into a physical address.

1. We find a physical address value for the binding
2. We find a hint to where the object currently resides

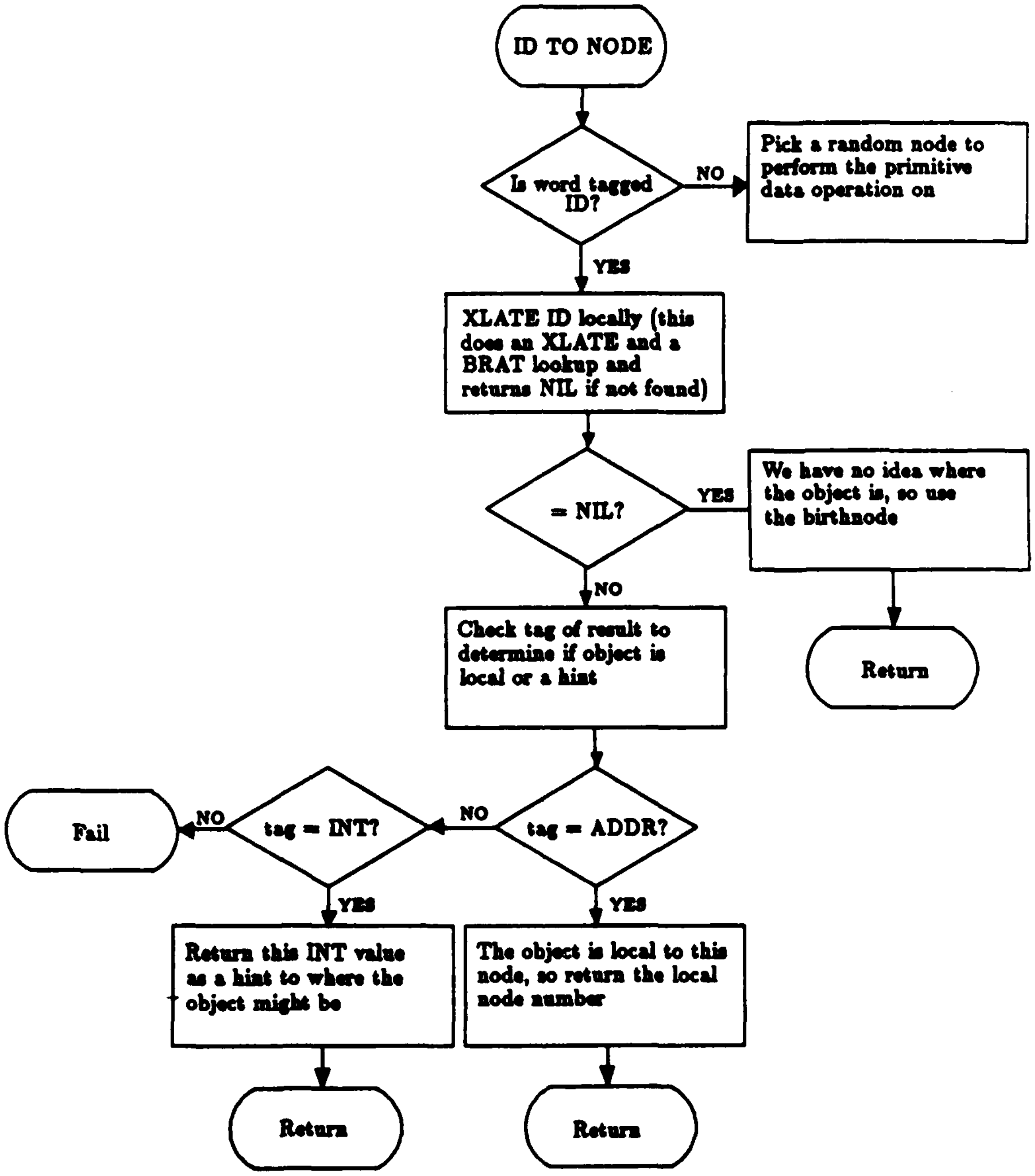

ID TO NODE
Is word tagged ID?
NO
Pick a random node to perform the primitive data operation on
YES
XLATE ID locally (this does an XLATE and a BRAT lookup and returns NIL if not found)
= NIL?
YES
We have no idea where the object is, so use the birthnode
Return
NO
Check tag of result to determine if object is local or a hint
tag = ADDR?
NO
tag = INT?
NO
Fail
YES
Return this INT value as a hint to where the object might be
Return
YES
The object is local to this node, so return the local node number
Return

Figure 4.3: Flowchart for the ID_TO_NODE algorithm

3. We find no binding for the object

Case 1 is the normal situation. The physical address associated with the object ID is returned. Case 2 implies that the object is rumored to be on a foreign node. We then send a request to this node asking that the object be shipped here for processing, and we suspend our process onto a wait list. Case 3 occurs when a node has no idea where an object resides. In this case, we send a request to the birthnode asking for the object. If the birthnode doesn't know where an object is, it loops, mailing messages to itself, assuming the object is in a state of transition somewhere.

## 4.5 Bouncing Objects

Note that this method of finding data objects may cause them to bounce around from node to node, as different processors wish to compute on them. This is the direct result of several design decisions: (1) each processor executes only one task at a time, (2) memory is not shared among processors, (3) mutable data objects are not cached, and (4) an object's data lies entirely on one node. The first and second decisions are fundamental to the design of our machine. We chose the grain size and memory model to provided a moderately fine grain, highly scalable processor. We chose not to do object caching because it is expensive to do in software, and is difficult on a network based memory model. It may be possible to provide coherent caching in the future however. The final restriction, that an object's state is contained on one node only is for simplicity's sake, and can be at least partially lifted by the introduction of "distributed objects" described in a later section.

So, with these characteristics in mind, it becomes important for us to try to prevent unnecessary "pinging" of objects from node to node. One way this is done is by "sending work to the object" rather than "sending the object to the work". Unfortunately, this is difficult to do in the general case due to problems with transferring processor state. As a compromise, we set the following policy.

1. If we were sending a selector to an object, and the object is not local, we forward the selector to the location of the object[1].
2. If we were accessing a non-local, immutable object, we halt, saving our process state, request a copy of the object, and restart execution when the copy arrives.
3. If we were accessing a non-local, mutable object, we halt, saving our process state, move the object here, and restart when it arrives.

This policy reduces the severity of the "pinging" problem, because work tends to accumulate at the object, while at the same time, allowing the object to move if it has to.

## 4.6 Details About Object Migration

This section formalizes the mechanisms provided to migrate objects. When we try to access a non-local object, we mail away to request a copy of the object or to move the object (depending on whether the object is immutable or mutable, respectively)[2]. When we wish to request a non-local object, the following steps are taken:

1. The processor state is saved in a context object, and the context is marked waiting for the ID of the object being requested.
2. The context is placed in a *resource wait table* that indicates processes waiting on objects.
3. A MIGRATE_OBJECT message is sent to the best guess residence of the object, asking it to be migrated to the requesting node, and the process suspends, able to execute the next message in the queue.
4. This MIGRATE_OBJECT message is forwarded down the chain of hints. If it lands on a node with no binding for the ID in question, the search continues at the birthnode. Finally this message arrives at the node the object resides on, and the message handler is run.
5. If the object in question is marked *unmovable*, then the message is sent back to the start of the queue, otherwise the message handler decides whether the object is mutable or not, and acts depending.
   - If it is mutable, the bindings are removed from this node, the object is mailed in an IMMIGRATE_OBJECT message back to the requesting node, and the object is deleted.

[1]The class/selector late-binding activation model is discussed in detail in chapter 6.

[2]Since a process cannot be interrupted by a same priority message, it does not suffer from livelock and can always make headway.

   - If the object is read-only, the data is mailed in an IMMIGRATE_COPY message back to the requesting node.

6. These messages eventually arrive back at the requesting node.
   - When a IMMIGRATE_OBJECT message arrives, the message handler (1) allocates the object, (2) marks the object unmovable (until it can update the birthnode, to prevent a race condition where hint updates may occur out of sequence), (3) copies the data into the object, (4) mails a NOW_RESIDING_AT message to the previous node of residence, and (5) calls the RESOURCE_ARRIVED system call, which will queue the restart of the waiting contexts.
   - When a IMMIGRATE_COPY message arrives, the handler (1) allocates the object, (2) marks the object header as a copy, (3) binds the old ID to this new object, (4) copies the data into the object, and (5) calls the RESOURCE_ARRIVED system call, which will queue the restart of the waiting contexts (copies can be collected when storage runs low).
7. The NOW_RESIDING_AT message makes a hint from the current node to the new node, and mails a UPDATE_BIRTHNODE message to the birthnode of the object, telling it of the object's new location.
8. The UPDATE_BIRTHNODE message makes a hint to the new location and mails an OBJECT_MOVABLE message to the location of the new object, passing its ID.
9. The OBJECT_MOVABLE message marks the object movable. Now the object is free to move again.

Figure 4.4 shows an example of this process.

## 4.7 Summary

The addition of a mechanism for object migration adds much more flexibility to the Jellybean system. Without imposing policy, the migration and copying system provides the basic mechanism for resource sharing. To alleviate name resolution bottlenecks at object birthnode, I designed a system of cycle-free hints to indicate where objects currently lie. It is not clear how long to allow these chains of hints to be. Long chains of hints would cause unnecessary network traffic and increase latency. Having single hints would increase the number of birthnode accesses and require mechanisms for removing old links. The system currently supports chains of hints.

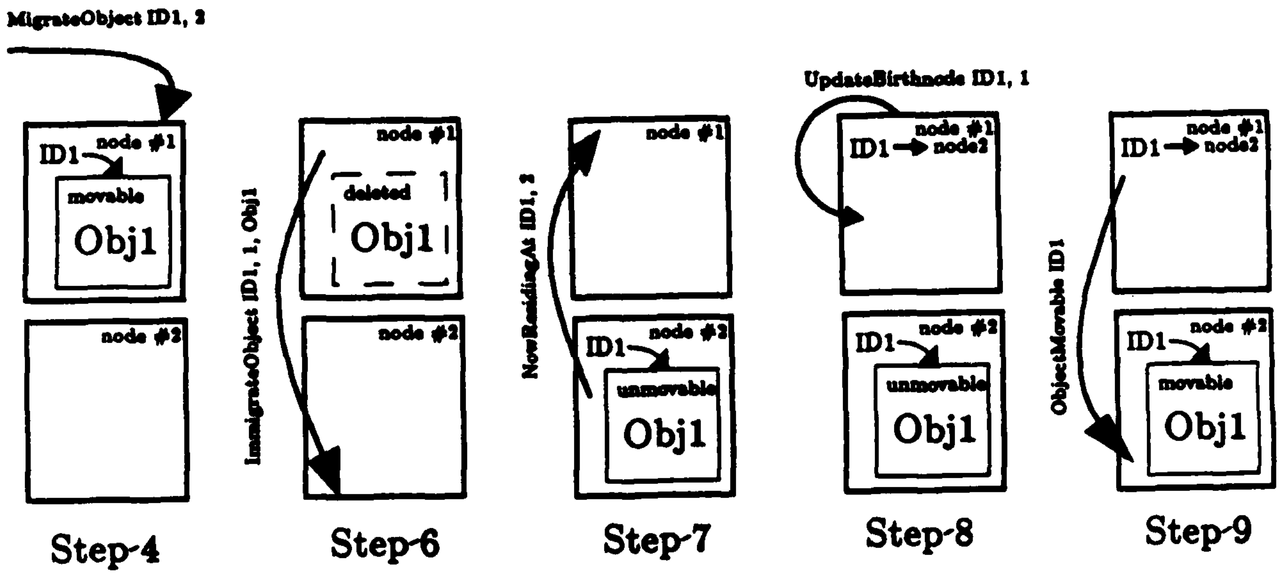


Figure 4.4: Step-by-step Object Migration

# Chapter 5

# A Virtually Addressed Code Execution Model

*They shall mount up with wings as eagles;*
*they shall run, and not be weary, and*
*they shall walk, and not faint*

— *The Holy Bible, Isaiah, 40:31*

At the most primitive level, we could execute physically addressed blocks of machine code by directly setting the registers, or by sending primitive messages. Unfortunately, we have no mechanism to allocate or relocate these blocks of code, they are physically addressed and sedentary. This chapter presents the system mechanisms that interact to provide a more flexible, but low overhead model for code execution by taking advantage of the virtually-addressed, object-based storage model we developed in the last 2 chapters.

I will present (1) the advantages of an object-based code model, (2) the mechanisms for executing object-based code, (3) local caching of methods, (4) contexts, suspension, and waiting for resources, and (5) efficient ways of distributing code models across a large network.

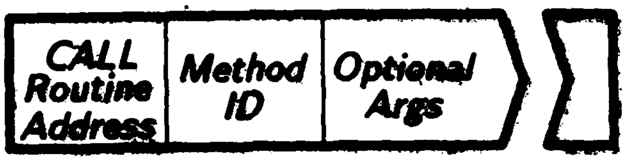


Figure 5.1: Format of the CALL Message

## 5.1 Taking Advantage of Object Storage

By taking advantage of the object storage and naming system we developed, we are able to wrap threads of code inside objects and gain all of the benefits of this more powerful object-based abstraction, of which a few are: (1) dynamic allocation, (2) relocation, even across nodes, and (3) convenient naming and name resolution. This view of code blocks as objects (or *methods*, which is what we call code blocks that are wrapped in objects) allows us to consider more advanced calling models, such as the ability to conveniently support remote procedure calls (RPCs) and the flexibility to "send the work to the data" rather than just the typical mechanism of "bringing the data to the work".

## 5.2 An Overview of the CALL Message

Ignoring for the moment the question of initially creating methods, let's concentrate on the mechanisms needed to execute them. The operating system provides a primitive message handler for a CALL message. To start a method running, we mail a CALL message to the node the method resides on[1], passing as arguments the virtual ID of the method to execute,

[1]Since we build this on top of the virtual, distributed namespace model, we can use hints to make our best guess where method resides.

and any data the method expects as parameters. The format of the CALL mesage is shown in figure 5.1. When the CALL message arrives at the node it first checks if the method is here. If so, the code is started. If not, rather than forward the message to the birthnode, we note that

1. Methods are immutable, and therefore can be copied
2. Certain methods might tend to be called often from many nodes

and adopt a policy of copying the method to this node. This way we provide local copies on many nodes (these can be periodically purged by some appropriate stategy to free up memory).

Once the method is on the node where the CALL message arrived, the message can start up the method. It does that by

- Translating the ID of the method into its physical address
- Placing this physical address of the code block in A0[2]
- Placing a 2 in the IP register

These steps will start the processor executing instructions from the method, starting at the third word. We skip the first two words of the method, because these hold object header information. The steps of the CALL message are schematically charted in figure 5.2. If the method somehow relocates on us while we were executing[3], the process that relocated the object will invalidate the A0 register. When our process starts again, it will fetch an instruction through A0 and cause an *invalid address* fault. This will run an exception handler to retranslate the method ID (in ID0) into the physical address (putting it in A0 again), and we will continue as if nothing had happened.

[2]A0 always points the the base of the code currently executed, unless the processor is in *absolute mode*, where this value is treated always as 0, regardless what it holds. The IP register holds the relative offset of the program counter within this code block starting at A0. (If we are in absolute mode, the IP register acts in effect like an absolute address rather than a relative address, because absolute mode makes the processor pretend the value of A0 is 0.)

[3]This could be caused by heap compaction, or the method being migrated to another node to free up space, among other reasons

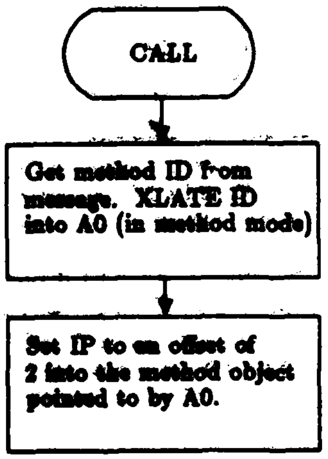

CALL
Get method ID from message. XLATE ID into A0 (in method mode)
Set IP to an offset of 2 into the method object pointed to by A0.

Figure 5.2: Flowchart of the CALL Message Handler

## 5.3 Caching Method Copies

Since method code is immutable, we can cache methods, just as we can cache other read-only data. To request a copy of a method we:

1. Allocate a context object to hold our processor state, so we can restart later
2. Copy the processor state into the context
3. Place the context in the *resource wait table* indicating that our context is waiting on this requested method
4. Mail off, requesting a copy of the method
5. When the method arrives, it is placed on our node and our context is restarted

These cached copies will have the *copy bit* set in the object header so that the storage reclaimer will know that this cached object is a duplicate, and can be purged if space is tight. Let's now look in a bit more detail at contexts and this resource wait table, two crucial mechanisms for supporting high level execution control.

## 5.4 Contexts

### 5.4.1 Why Do We Need Them?

Contexts are just objects that hold the important state of the processor, so the current task cab be halted and later restarted where it left off. In addition, contexts can provide space for local variables used in the task's computation.

### 5.4.2 How Do We Make Them?

Contexts are allocated by the NEW_CONTEXT system call. The call takes as an argument, the number of additional variables needed, and it returns a context big enough to hold the minimum necessary processor state plus the additional variables. When a process is done

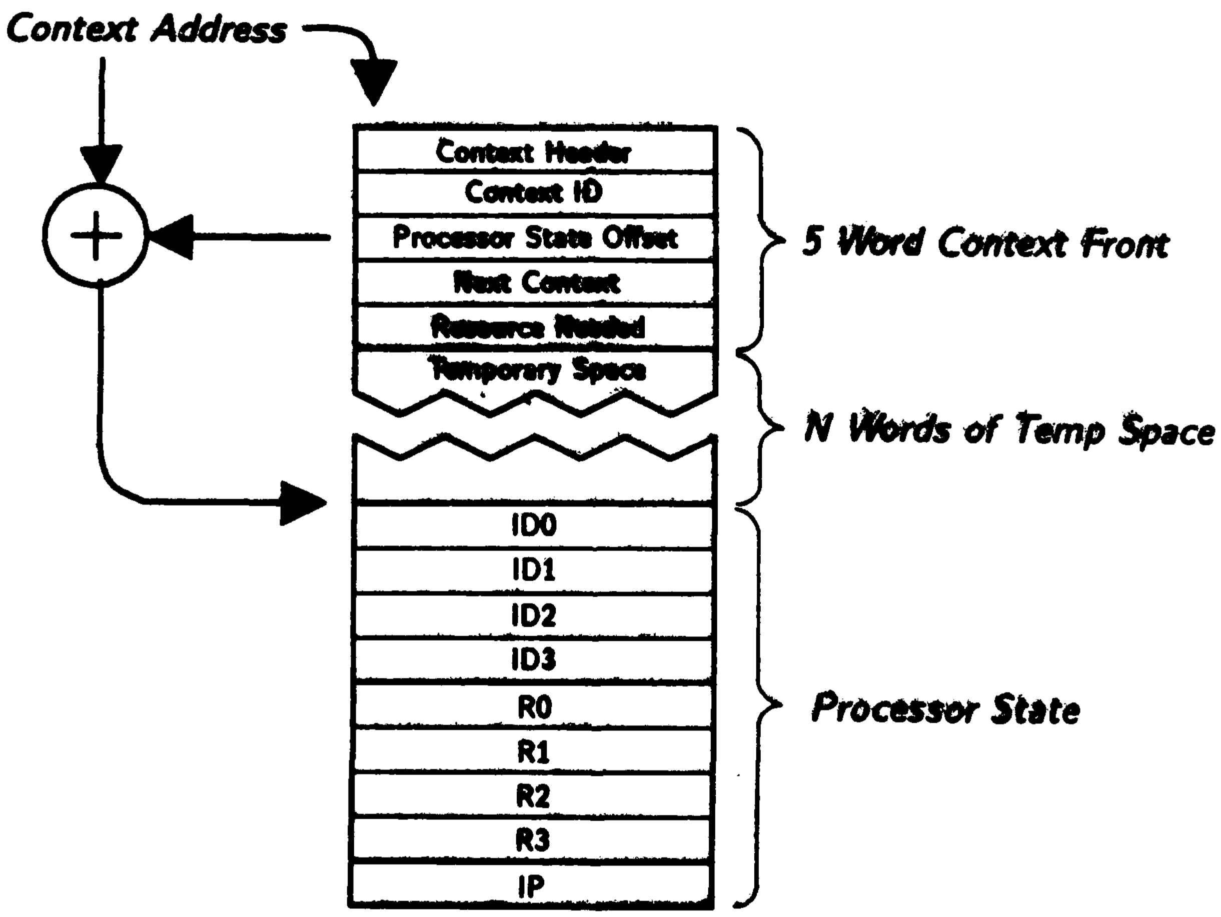


Figure 5.3: Structure of a Typical Context

with a context, it should explicitly deallocate it with the FREE_CONTEXT system call. Figure 5.3 shows the format of a typical context.

As with all objects, the first two words are used by the object manager. The next three words are used to hold an offset to the processor state part of the context (for faster restarts), a pointer to the next context in a list of contexts, and a value indicating that the context is waiting on a particular resource. The context then contains some amount of user reserved space follwed by nine words of processor state. The minimal size of a context, with no user space is 14 words.

### 5.4.3 How Do We Make Them ... Quickly!?

Since we expect contexts to be used very often, and since we want method startup costs to be small and methods to be short, we don't want a majority of our execution time to be spent allocating contexts. To accomodate these constraints, we reuse old contexts rather than allocating new ones each time. When a context is deallocated, it is placed back on a *free context list*. The next time a context is requested, we try to re-use one from the free list, since this will take only a few instructions.

However, contexts vary in size, and we wouldn't want to have to walk the list each time to see if we have a context big enough to meet our request. So, we only save contexts that meet a common size. This way, any time we request a context of this "common" size, we can yank the first one off of the free list and use it. The format of the free context list is shown in figure 5.4.

The first context in the free context list is pointed to by the CONTEXT_FREE_-LIST operating system variable. If no contexts are in the free list, the OS variable is set to NIL. Each context in the free list points to the next context in the list by the context's NEXT_CONTEXT slot as shown previously in figure 5.3. The final context in the free list has its NEXT_CONTEXT slot set to NIL.

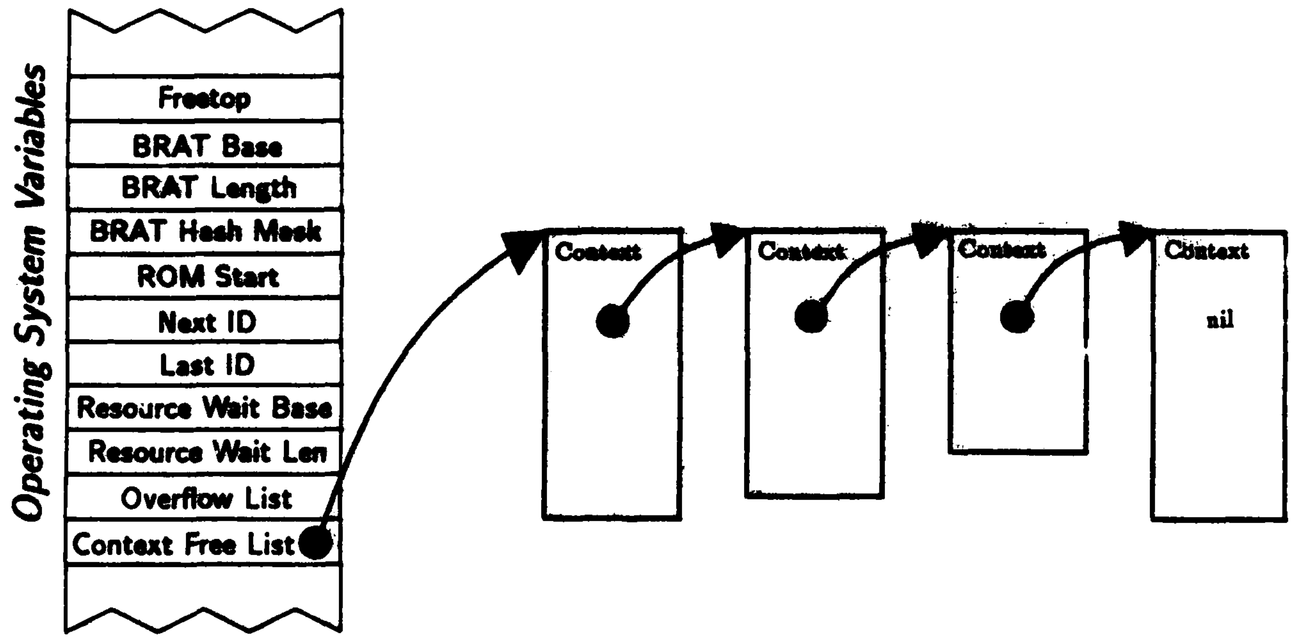


Figure 5.4: The Free Context List

### 5.4.4 Restarting a Context

The operating system provides one primitive message (RESTART_CONTEXT) and two system calls (XFER_ID and XFER_ADDR) to restart a context. The system calls take either an ID or a physical address of a context, and restarts it, copying the processor state from the context to the processor registers. The restart context message takes a context ID and transfers control to it by calling the XFER_ID system call on the context ID.

## 5.5 The Resource Wait Table

The *resource wait table* is a system data structure that indicates which contexts are waiting for which services. It consists of two parts. The first part of the wait table is a fixed size associative table that binds resource IDs to waiting contexts. Figure 5.5 shows a portion of a hypothetical table. We see several contexts waiting for ID1, one context waiting for ID2, and the rest of the slots are empty. Empty slots are set to NIL. When a resource arrives, the wait table is searched, and the contexts in the list bound to the ID are restarted.

Searching this table is fast, but unfortunately, we can not bound the number of entries that try to occupy the table. At some time, we may run out of room. When this happens, we resort to a slower form of data structure and link the contexts waiting on resources in a list called the *resource overflow list*. If we don't find a binding in the table, we begin searching the list of contexts. Since each context has a RESOURCE_NEEDED slot, we can always tell what resource the context is waiting for. This provides us a way to continue if the table becomes full. By sizing the table appropriately, it may be possible to limit use of the overflow list to a minimum.

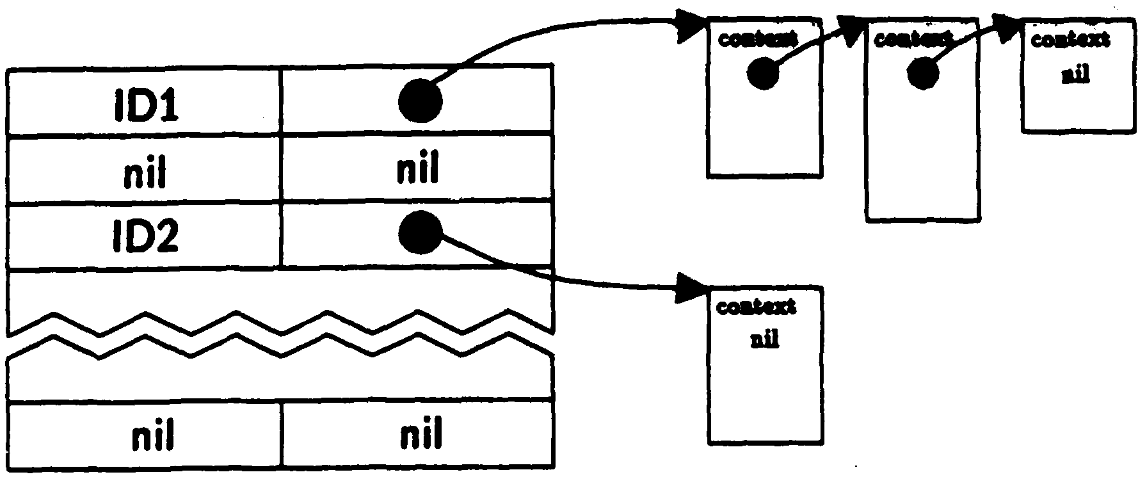


Figure 5.5: The Resource Wait Table

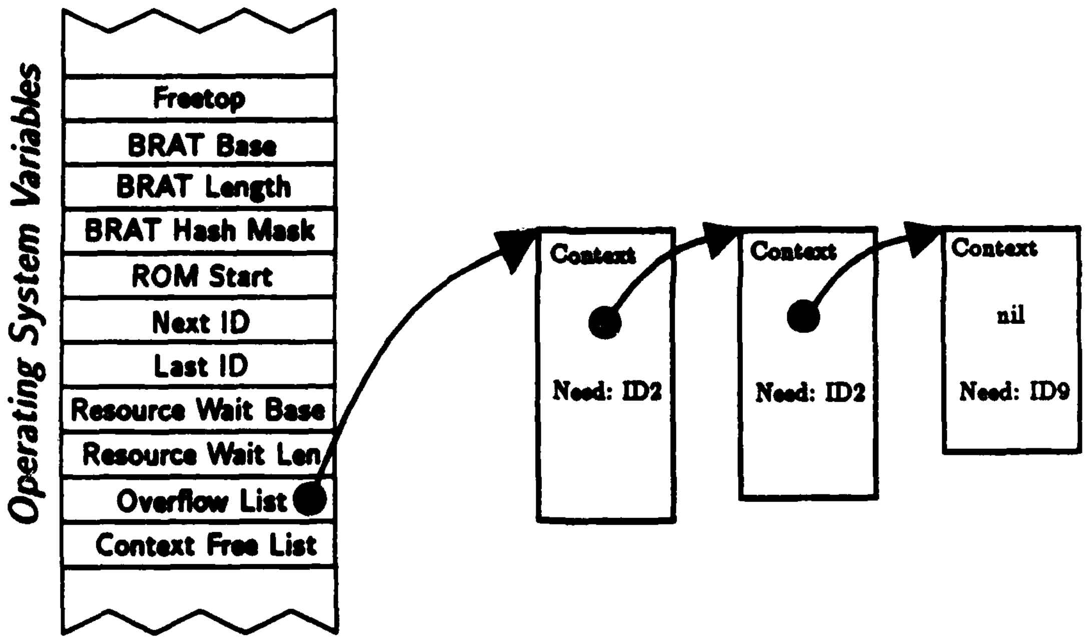


Figure 5.6: The Resource Wait Overflow List

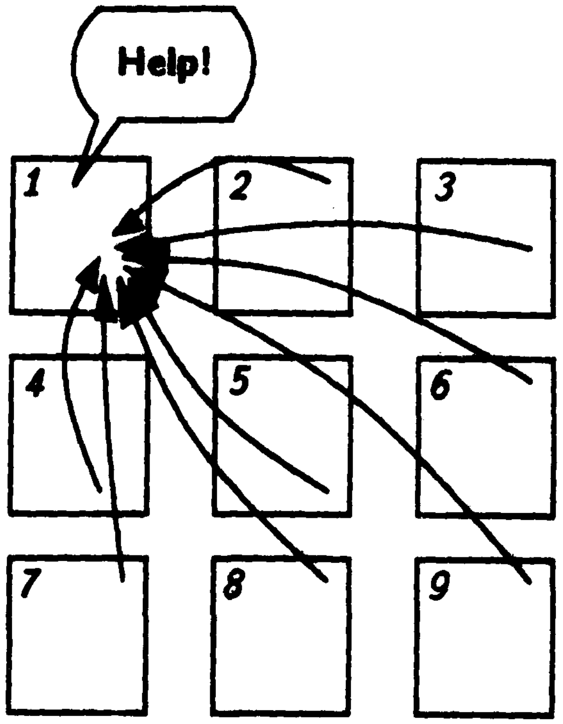


Figure 5.7: A Parallel Resource Request Bottleneck in a 3 x 3 Network

## 5.6 Removing Method Caching Bottlenecks with Distribution Trees

The current scheme for method caching implies that in many cases, nodes wanting methods will have to ask the birthnode of the method (or at least the residence node) for a copy. If many nodes simultaneously need the same method (as will likely happen with highly parallel execution), then the birthnode will be deluged with method requests which it can only handle sequentially. These bottlenecks could degrade performance considerably. For example, figure 5.7 shows a network of 9 processing nodes. Suppose nodes 2 - 9 all requested a method copy from node 1. Node 1 would receive a barrage of 8 requests for the method which would eliminate all parallelism, since it could consider each request only sequentially.

One way to reduce the threat of performance degrading bottlenecks is to set up a *distribution hierarchy*, so that each node requests resources from its local distribution center (the distribution hierarchies are different for different resources). Each of these local centers would make requests to its superior, all the way up to the master resource center. We can use this type of distribution graph to help in requesting method copies (or copies of any type of immutable data for that matter).

Take again the 3 x 3 node network example, where 8 nodes request a method from node 1, but this time impose a distribution bureaucracy like that shown in the tree in figure 5.8. This time, node 1 only has to handle 3 messages, from nodes 2, 4 and 5. Each of these nodes serve as local distribution centers for the remaining nodes. Node 2 services nodes 3 and 6, node 4 services nodes 7 and 8, and node 5 services node 9. In this manner we have permitted more parallelism to continue, as well as limiting the burden on node 1 (which could cause queue overflow, network blocking, and other conditions where performance degrades considerably).

Let's now discuss some ways that a distribution tree method caching scheme can be implemented in the Jellybean Machine system software. First, what are the contraints we are working under?

- The distribution tree edges must be easily computable
- We need to make reasonable choices for *branching factor* versus *tree depth*. Too high a branching factor might create bottlenecks, but too low a branching factor would tend to cache unnecessary copies, and suffer long latency as the birthnode was many edges away from the requesting node.
- We would like to have significantly different trees for different resources. Different methods should have different distribution hierarchies, again to decrease bottlenecks, and to distribute resources more thoroughly.

One fairly simple first attempt at a distribution tree formula might be to go to the distribution center that is halfway between the current node and the birthnode in terms

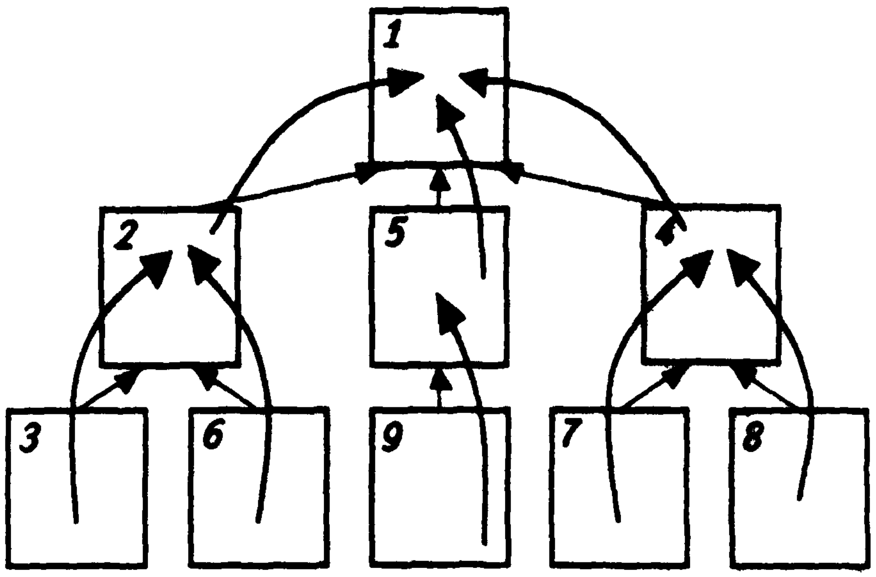


Figure 5.8: A Distribution Tree Bureaucracy To Balance Load in a 3 x 3 Network

of hops. In other words, to find the next regional distribution center, given the birthnode coordinates $(x_b, y_b)$ and our current coordinates at $(x_c, y_c)$, we would calculate the halfway coordinates $(x_{\frac{1}{2}}, y_{\frac{1}{2}})$ by:

$$\begin{aligned}
\Delta x_{\text{real}} &= \frac{x_b - x_c}{2} \\
\Delta y_{\text{real}} &= \frac{y_b - y_c}{2} \\
\Delta x &= \begin{cases} \lceil x_{\text{real}} \rceil & \text{if } \operatorname{sgn} x_{\text{real}} \geq 0 \\ -\lceil |x_{\text{real}}| \rceil & \text{if } \operatorname{sgn} x_{\text{real}} < 0 \end{cases} \\
\Delta y &= \begin{cases} \lceil y_{\text{real}} \rceil & \text{if } \operatorname{sgn} y_{\text{real}} \geq 0 \\ -\lceil |y_{\text{real}}| \rceil & \text{if } \operatorname{sgn} y_{\text{real}} < 0 \end{cases} \\
x_{\frac{1}{2}} &= \lceil x_c + \Delta x \rceil \\
y_{\frac{1}{2}} &= \lceil y_c + \Delta y \rceil
\end{aligned}$$

This is in fact the algorithm used to create the distribution tree in figure 5.8. Figure 5.9 shows several distribution trees created by this algorithm for networks of various sizes and various birthnodes. This method creates trees with depth at most $\log_2 m + 1$ for a network with a maximum dimension of $m$ nodes. So, for a reasonable sized machine of 4096 nodes (64 x 64) we would at most have to traverse $\log_2 64 + 1$ or 7 edges of the distribution tree. For enormous systems, say 1K nodes on a side, the tree depth will be only 11.

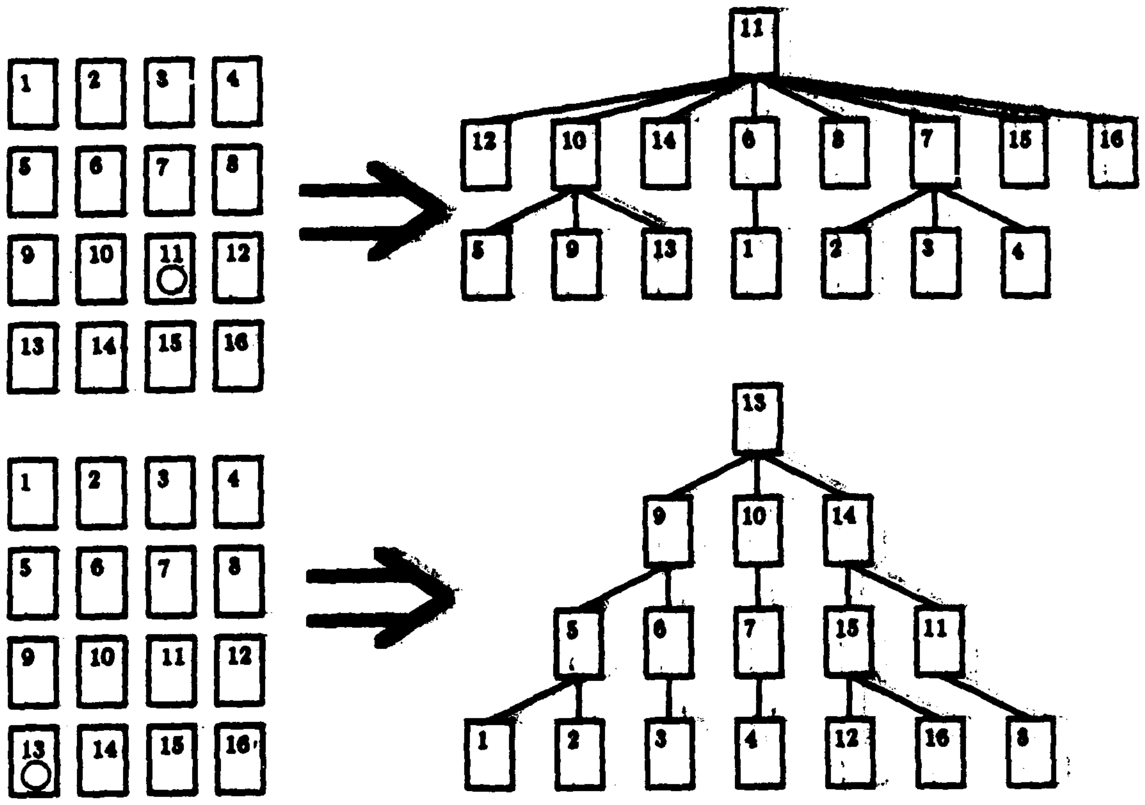


Figure 5.9: Example Distribution Trees for Several Machine Configuration

# Chapter 6

# System Support of a Type-Dispatched Calling Model

*We never sent a messenger save with
the language of his folk, that he
might make the message clear for them*

— *The Koran, 13:11*

One of the most important aims of the Jellybean Machine is to provide a concurrent processor that efficiently supports object-oriented, late-binding procedure activations. This chapter introduces the idea of message-passing and late-binding programming methodologies, and discusses the system services in the Jellybean Machine operating system that support this manner of programming.

## 6.1 Message-Passing and Object-Oriented Languages

There has been much interest during the past few years in "object-oriented" programming. Though this term is not particularly precise, it does describe a fairly cohesive set of languages exhibiting behavior markedly different from the typical Algol-like programming style. There are two characteristics in particular that languages typically categorized as object-oriented share.

First of all, operations tend not to be thought of as functions applied to data objects, as they are in Algol derivatives. Instead, data objects are "personified" as "actors" that receive requests made of them. These requests are made by "sending a message" to an object called the *receiver* of the message. The operation that was requested of the object is typically called the *selector*, since it selects the object to be performed. So, where a standard language Algol-like language might calculate the determinant of a matrix m by

```
determinant(m);
```

and object oriented implementation might look something like

```
(send m 'determinant)
```

We call this concept of performing operations by sending selectors to objects the *message-passing paradigm*. This paradigm turns out to be a very convenient model of computation.

The second characteristic of object-oriented languages that make them appealing is the fact that the operations on different data-types can have the same names. This allows us, for example, to have an 'area selector for circle data types, as well as an 'area selector for polygon data types. In many other languages this would cause a naming conflict, requiring us to set up an explicit naming convention, such as calling circle_area() and polygon_area() routines on objects of the proper type.

But, more importantly than just saving us the hassle of naming conflicts, object-oriented languages actually decide which procedure to run for a certain data type. In other words, when an 'area selector arrived at an object, the system would decide whether this object is a circle or a polygon and automatically run the correct procedure. In addition, if the receiver of the 'area selector was not a data type that supported the area operation (such as an integer), then an error would be reported by the system. In Algol-like languages, it is the burden of the programmer to know the type of the object he is dealing with, so he can call the proper operation. This is crucial in many symbolic languages with loose type-checking, like Lisp, where we can have lists of many different types of objects[1]. This is called a *late-binding activation* since we don't decide what routine will be run at compile-time, but instead wait until later, when the message send is actually done.

Operations with the same name and semantically similar meaning supported by various data types are called *generic operations* since these operations represent the generic behavior the programmer wants to accomplish (add things, draw things, calculate areas of things). The *specific* behavior is calculated at run-time once we know the data type of the object (called the *class* of the object), and the selected operation, by a process known as *class-selector lookup*.

So, object-oriented languages have two main components

1. Procedures are activated by the *message-passing paradigm* rather than a more applicative model of programming.
2. Each data type has its own set of supported operations, where names can be the same as in other data types, and may represent *generic operations* over varied data types. Activations are caused by *late-binding sends* which lookup the *specific operation* to run based on the class of the object receiving the message (the *receiver*) and the selected operation (the *selector*).

Our goal now is to provide a system substrate that will efficiently and conveniently support these aims.

[1]A good example of this is an object oriented drawing program, where we have a list of many different types of objects that are in the current picture. A convenient way to refresh the screen in an object-oriented system is to send a 'draw message to each object in the list. Based on the data type of each object at run-time, the appropriate routine (circle draw, rectangle draw, text draw, etc.) is activated

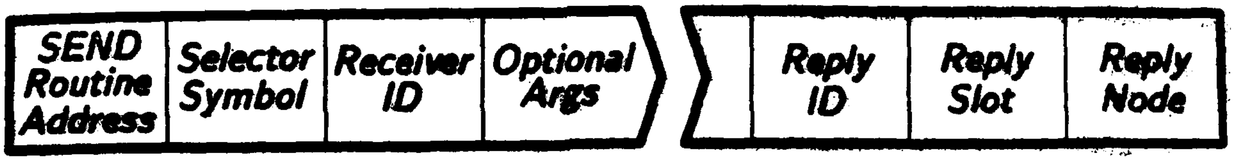


Figure 6.1: Format of the SEND Message

## 6.2 Late-Binding Send Execution Support

The next task of the operating system is to provide a mechanism to simulate the message-passing paradigm. We already have network communication hardware that allows data to be sent between nodes. We also have a global object namespace provided by the virtual memory extensions. Together, we can use these components to implement the message-passing execution model.

To do this, we implement one more primitive message, the SEND message handler (not to be confused with the SEND machine instruction). This primitive message handler acts in the object-oriented manner we showed earlier. Figure 6.1 shows the significance of the different words of the message. The first word is the address of the SEND message handler, the second word is the selector, the third word is the receiver. The rest of the words are arguments, and information about where to reply to.

When the SEND message arrives on the node that the receiver resides on (we forward this SEND message to wherever the receiver resides) the primitive message handler is started. Figure 6.2 shows a flow chart that describes how the SEND message handler works. It first picks the class our of the receiver object (so we know what data type the receiver is). We then merge the class and selector together into a class/selector word (shown in figure 6.3). Now that we have the class and selector, we try to see if there is a class/selector →

method ID binding in the cache. If so, we start the method with the CALL message as discussed in the previous chapter. If not, we need to lookup the binding.

At the current time, we do not have enough insight into the characteristics of machine behavior, to feel comfortable locking down the class/selector lookup algorithm. For this reason, we provide the lookup routine in a method. We insist that this method is allocated before any others so it always has the same method ID. This LookupMethod method takes the class and selector, and consults some distributed system table to find the method ID corresponding to this class and selector.

## 6.3 Loading Class/Selector Methods into the System

Let's now briefly look at how the class/selector method information is loaded into the Jellybean system. Figure 6.4 shows the schema for how the compiler and run-time environment will interact with the Jellybean Machine processing network. The compiler is responsible for generating class and selector numbers and for compiling the source language into MDP machine code. A certain node of the network is picked for the method to reside on by some distribution policy. The method data as well as the class and selector that this method represents are sent to this chosen node by the NEW_METHOD message. The format of a NEW_METHOD message is shown in figure 6.5.

When a NEW_METHOD message arrives at a node, the NEW_METHOD message handler begins executing. It makes an object to hold the method, and copies the code from the message into the object. The NEW_METHOD handler then calls the InstallMethod method which takes the class, selector, and method ID and makes the bindings in the class/selector → method ID data structures.

Specification of the class/selector → method ID data structures has been ignored without attempts at subtlety. We do not have enough insight to definitely specify the best

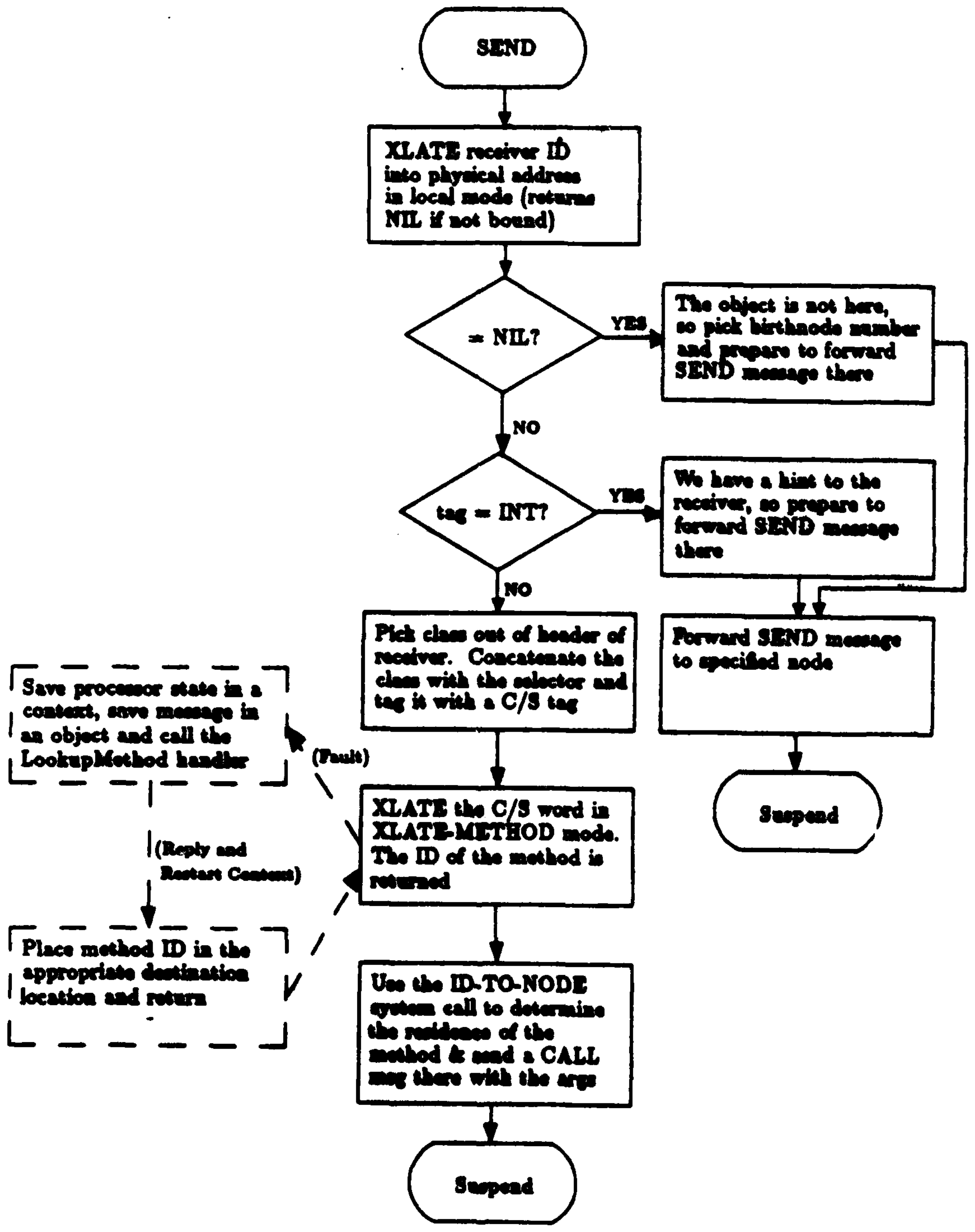

SEND
XLATE receiver ID into physical address in local mode (returns NIL if not bound)
= NIL?
YES
The object is not here, so pick birthnode number and prepare to forward SEND message there
NO
tag = INT?
YES
We have a hint to the receiver, so prepare to forward SEND message there
NO
Pick class out of header of receiver. Concatenate the class with the selector and tag it with a C/S tag
Forward SEND message to specified node
Suspend
Save processor state in a context, save message in an object and call the LookupMethod handler
(Fault)
XLATE the C/S word in XLATE-METHOD mode. The ID of the method is returned
(Reply and Restart Context)
Place method ID in the appropriate destination location and return
Use the ID-TO-NODE system call to determine the residence of the method & send a CALL msg there with the args
Suspend

Figure 6.2: Flowchart of the SEND Message Handler

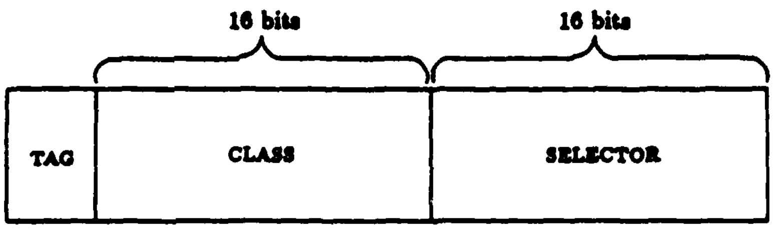


Figure 6.3: Class/Selector Word Format

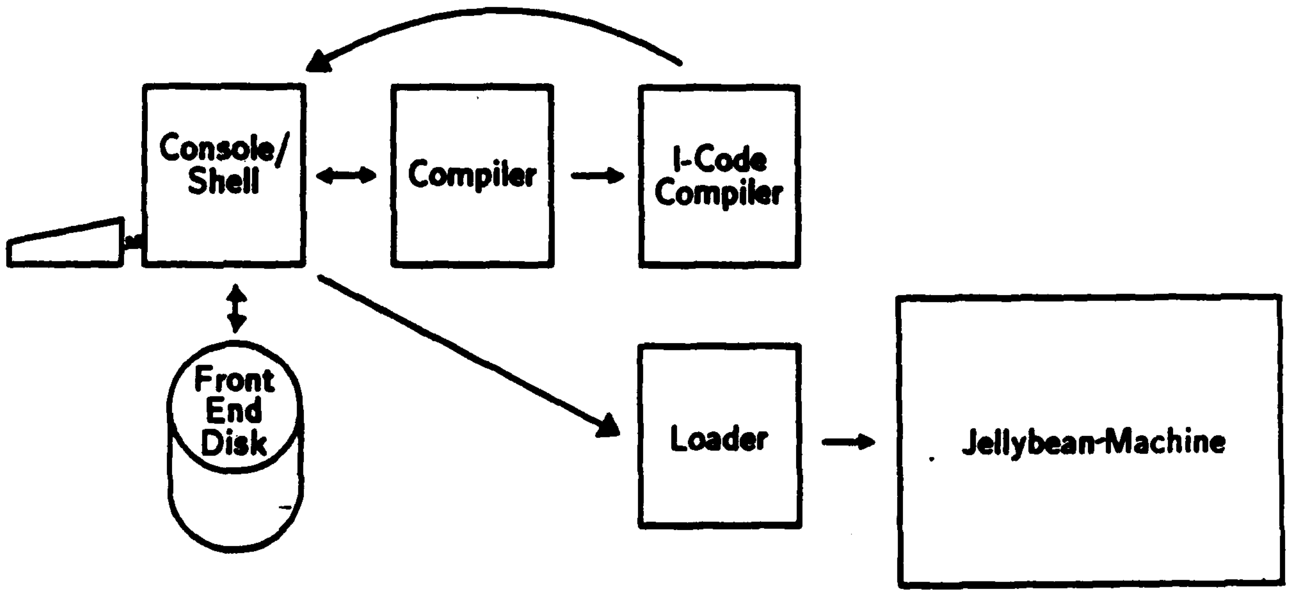


Figure 6.4: A Coarse View of the Compiler/Machine Interface

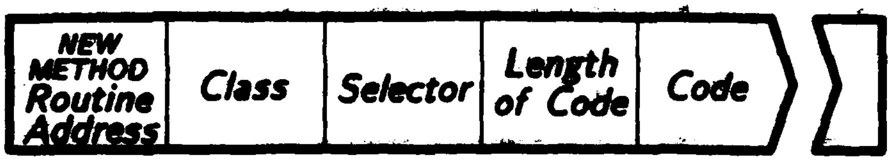


Figure 6.5: Format of the NEW_METHOD Message

format for these tables. We can talk a bit about the issues involved. (1) We should be able to take a class/selector word and efficiently find the corresponding method ID. (2) The table should be distributed around the network in a way to minimize bottlenecks.

A reasonable way of doing this would be to apply some "bit-twiddling" function to the class/selector words to decide what node is responsible for knowing their bindings. The actual data structures could be hashed, or perhaps each class would have an object that holds the method IDs for every selector. One annoying problem with any approach is the boot-strapping problem. We need to know how we can get to the data. Because of the added indirection through the LookupMethod and InstallMethod handlers we have the flexibility to try several approaches and test their performance in the future.

## 6.4 Returning Values

Return values can be sent with the REPLY message. This message takes the context ID to reply to, the slot number of the context to fill, and one word of reply data. The reply data is passed by value if it is a primitive data word, or by reference if an object is to be returned.

## 6.5 Summary

The class/selector calling model is a convenient mechanism for invoking tasks. By implementing it efficiently in the operating system kernel, we can guarantee an efficient implementation. To provided extensibility, we provide hooks to the LookupMethod and InsertMethod handlers, so these routines can be reconfigured independently of the rest of the kernel.

# Chapter 7

# Storage Reclamation in the Jellybean Machine

*But virtue, as it never will be moved,*
*Though lewdness court it in a shape of heaven,*
*So lust, though to a radiant angel linked,*
*Will sate itself in a celestial bed,*
*And prey on garbage*

— SHAKESPEARE, in *Hamlet I, V. 53*

## 7.1 Introduction

The successful performance of our machine relies on the fact that sufficient parallelism exists on the grain of methods. In order for this to happen, it is important that data-dependencies to shared objects are minimized, by adopting a more functional approach, where methods interact by value rather than by reference, as much as possible. This situation promotes a large number of small, short-lived objects. Because of the minute amount of memory per each processing node, an efficient storage reclamation mechanism becomes an important facet. The characteristics of our system, however, cause many straightforward methods of storage management to break down. In this discussion we will examine some of the important properties of the Jellybean Machine, and the ways these properties influence reclamation. The rest of this chapter provides a discussion of the issues pertaining to reclamation on the Jellybean Machine, and a possible first-cut at a garbage collection algorithm.

## 7.2 Automatic Collection is Desirable

Because the system is object oriented, and because we have a small memory with frequent allocations, object reclamation is important. Because objects can be shared in complex ways, and because of the high level programming model we wish to support, we wish most object deallocations to be handled automatically by a "garbage collector" that searches for objects that are no longer in use (i.e. there are no pointers to the object anywhere) and deallocates them when necessary.

## 7.3 Choosing a Collection Approach

Several characteristics of the Jellybean Machine will guide us in the choice of garbage collection. Let's remind ourselves of the character of the machine.

### 7.3.1 Memory Organization

The memory in a Jellybean processor is small, and it is local to that processor. Memory allocation is done in a simple contiguous manner. Compaction can be done in parallel very quickly. Memory objects are segment-based and are given unique object id's. In addition, these object id's are concatenated with a birth node number to provide a global virtual address. The virtual to physical translation mechanism uses caching to improve name resolution, but this relies on locality. Random access to many addresses could be very expensive.

### 7.3.2 Addressing System and Network Topology

The Jellybean Machine uses a distributed memory to provide "site autonomy" [LS80] in order to perform local operations very fast, and avoid memory conflicts. But, the tradeoff is that foreign accesses will be very costly, involving a message send mechanism that is at least an order of magnitude slower. In addition, distributed memory can require synchronization, and the delays of network communication may make certain synchronization conditions impossible. The network may cause bottlenecks to occur if too many messages are sent to one place, and may hold data in transit. The network latency may also be a factor.

### 7.3.3 Garbage Collection Character

Garbage collectors take on various different characters. The common approach of *reference counting* collection doesn't appear to be feasable in the Jellybean Machine because (1) it cannot collect cyclic data structures, (2) every pointer change will require a (possibly remote) object access, and (3) we are not always aware when "dead" pointers get changed. For these reasons, we decided to attempt some variant of a *pointer chasing* garbage collection mechanism. The next section describes the implementation of a pointer chasing garbage collector for our machine in some detail.

## 7.4 A Pointer Chasing Garbage Collector

There are several properties that we would like our garbage collector to have.

- The collector should be efficient in terms of time and message sends. We do not want the queues of all nodes to overflow with collection messages.
- The collector should run in the background or incrementally, for two reasons. First, we wish to take advantage of processor idle time so that we can squeeze as much computation out of our processor as possible. Secondly, we would like to avoid the situation where our machine runs for a while and then "hangs up" for an hour while garbage collection occurs.

### 7.4.1 The General Idea

Most of the work of pointer chasing garbage collection algorithms to date are targeted at sequential or shared-memory machines with large virtual memories. The standard algorithm is based on the copying collector proposed by Baker. This has been expanded into incremental collectors and has been tuned to various object lifespans, with a good degree of success. Still, these approaches are targeted at a genre of machine of a radically different character that the J-Machine. With an admitted scarcity of knowledge in distributed collection, the rest of this chapter serves only to sketch a simple vision of such a collector [Tot88], and some of the problems that are faced.

A simple collector would involve recursive marking by message sends, and would compact the heap rather than by scavenging or copying, due to the small amount of memory per chip. The phases of this simple collector would be:

**Desire** The desire phase occurs when some node or nodes has a desire to garbage collect. Perhaps a node or a certain number of nodes have run out of memory. Perhaps this occurs on a time count.

**Init** The initialization phase is where objects are marked *unreferenced* initially, as well as setting any necessary variables.

**Marking** The marking phase does a recursive descent of the reference tree starting at the root set, marking reachable objects with the *reachable* tag.

**Sweeping** When marking is done, the memory can be compacted by "sweeping" the good objects back toward the bottom of the heap, and changing their virtual $\rightarrow$ physical bindings.

### 7.4.2 Problems

#### Synchronization and "Travelling References"

A major problem in garbage collection across a communication medium is lack of synchronized, instantaneous transmission. This shows itself in garbage collection in a few ways. One of the more annoying problems is how to be sure that the last pointer to an object isn't in transit when the garbage collector comes along. The garbage collector doesn't see any pointers in the network, so an object may be deleted because a pointer was "travelling" between nodes where it can't be noticed. We can refer to this as the *travelling reference* problem. Figure 7.1 shows a portion of a network of processors, where an ID of an object is in the network when the collector is run.

An obvious way to resolve this situation is to prevent all upcoming message sends during collection, so that no other pointers are mailed into the network, and then to wait until all messages in transit have landed in a queue. We can tell when all messages have landed by either waiting a length of time we know to be longer than the maximum latency from the most distant nodes, or by sending "scout" or "bulldozer" messages down the network dimensions. When all these "bulldozer" messages arrive, they will have pushed all other messages out of the way, and the network will be empty.

#### Problems With Disabling Sends

In order to prevent the *travelling reference* problem, we have to

- Disable sends so no new references enter the network.
- Wait for all messages in the message in the network to land.

But, we have no explicit mechanism in the MDP processing node to disable sends[1]. If we did, we could allow the processors to run until they tried to execute one of these disabled

[1]Or more preferably - a mechanism that would disable any sends that would cause a reference to be mailed into the network - all other messages could continue

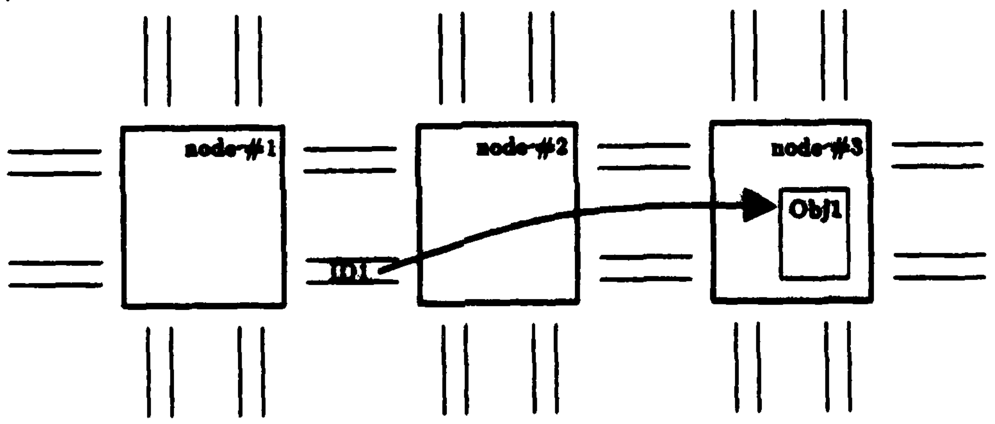


Figure 7.1: Object ID Travelling in Network

instructions. When this happened, a fault could occur and some manner of process halting could occur (such as saving a context for the process for later re-starting[2]).

A possible way to resolve this problem at first might be to place guards in certain high-level execution handlers such as SEND and CALL. These handlers are run when a SEND or CALL message (two messages that ask a node to start executing a method) arrives. Inside these handlers we could have a guard that would defer the execution of the method until collection finishes. This goes a long way toward resolving the problem of travelling references if most the code that mails IDs around is code that is executed with CALL and SEND[3]

Another way to shut down the machine might be to disable the queue execution. This would cause messages to back-up in the queues. Certain messages that we would want to execute could be done by having the processor "walking" the queue by hand looking for certain types of messages (such as garbage collection messages). It could also pull items out of the queue and into the heap to prevent queue overflow.

### Problems With Background Execution

Since, at the start of garbage collection, we stop message sends by various possible mechanisms, our concurrent machine is effectively shut down. This violates our desire for the collector to run in the background, in parallel with method execution.

[2]This, however, could lead to the difficult to resolve problem of insufficient memory for a context allocation. This might be likely since we are in the middle of collection. When there is not enough local memory, the standard mechanism is to do the allocation on a foreign node. But this requires mailing references in the network, which is exactly what we are trying to avoid. This underscores the difficulty present in providing efficient, convenient methods of prevent travelling references

[3]And this is likely to be true. Apart from CALL and SEND messages, all other messages are primitive system messages (where the system may have to be responsible for avoiding ID mailing during collection), and various other messages to create NEW objects and handle function returns. If we think of a CALL or a SEND as being a function call, then this guard method will eventually stop the machine, with every processor being idle or waiting to execute a function. This implementation has at least 2 requirements that we must always be aware of. (1) We must insure that all non-CALL and non-SEND messages must not violate the rules and mail references during garbage collection time. (2) Catastrophe can occur when we run out of memory trying to make contexts to hold the deferred execution requests.

In addition, the lack of a register set for background mode prevents any way for the Message Driven Processor to take advantage of idle time in a reasonable way. Since any message would take priority over background mode, the register set will be trashed. Any computation done in background mode must shut off interrupts, which instead of taking advantage of idle time, takes advantage of application execution time! Some compromises can be made, such as having background mode start up small units of computation by sending priority 0 messages, or by queuing up contexts of waiting-to-run background processes that are begun by a context startup message send when the background loop is entered. Again, various improvements should be examined.

## 7.5 Summary

The characteristics of the Jellybean machine necessitate a heap collector to reclaim storage. This collector may have to run often (since our nodes have such a small amount of memory). A reference counting approach seems to be out since there is a large overhead in changing the object reference counts (and it is difficult to know when a reference is written over and thus deleted) as well as the fact that it cannot handle cyclic structures (if we insist that cyclic structures are illegal that results in a big loss in terms of flexibility. If we don't collect structures, we will rapidly run out of memory). A pointer chasing collector has problems with *travelling references* (where the marker will not see the final reference to an object because it is in a network – and thus delete the object), but seems to be the most viable approach. It would be desirable to have the collector run in the background without shutting the machine down, but the travelling reference problem seems to make this difficult.

# Chapter 8

# Support for Concurrent Programming Languages

*I get by with a little help from my friends.*

— John Lennon and Paul McCartney, in *"A Little Help From My Friends"* (1967)

The Jellybean Machine Operating System Software provides several noteworthy services to support concurrent programming languages, both for functional and efficiency reasons. These include (1) the SEND and REPLY message handlers, (2) futures, (3) distributed objects, and (4) the interaction interface.

## 8.1 High-Level Languages

### 8.1.1 CST

Currently, the high-level language being used in the Jellybean Machine project is a Smalltalk-80 based language called CST (Concurrent SmallTalk) [DC]. CST uses a Lisp-like prefix syntax, and codes sends implicitly in a function application metaphor. CST allows asynchronous messages to exploit concurrency, and fully utilizes the late-binding execution model. Locks are provided for explicit synchronization, and a "distributed object" data type exists to scatter object state over a large area. This CST code will be compiled to intermediate code which will is passed through a back end that converts the i-code to MDP machine code and loads it into the system. The compilation and loading mechanism is was previously sketched in figure 6.4.

The rest of this chapter describes several operating system services that support the execution of the object-oriented model of computation.

## 8.2 SEND and REPLY

As discussed in earlier chapters, the SEND message handler provides the machinery to run a method based on the class of a receiving object and the selector symbol "sent" to the object. In the current system, the SEND message may also describe one object to return a value to. This *return-slot* is specified by passing the ID of the object to hold the returned value (the returned value must be one word, either a primitive value such as an integer or a symbol, or the ID pointer to the object), the *slot* (index into the object) number. and the node the object is on.

The REPLY handler actually performs the return of the value. The REPLY message mails the target object ID, the target variable number, and the one word return value to the node number specified in the SEND message. When a REPLY message arrives at a node, the returned value is stored in the indicated slot of the target object, and any processes waiting for a variable to be filled by a reply are restarted.

## 8.3 Futures

### 8.3.1 Conforming to Data Dependencies

Data dependencies impose an order on execution. If a computation result is used in a calculation, the result must be available before the calculation can occur. In a sequential processor, there is no problem. The instructions are ordered in such a way to insure that previous results are available in certain places before those values are needed. In a distributed processor, on the other hand, a computation may take an indeterminate amount of time to complete on a remote node. Because of this, we may get to a point where a value is needed before the calculation of the value has completed. It is necessary to wait until this result returns before continuing the calculation.

### 8.3.2 The Check's in the Mail

This section details a mechanism used prominently by the Jellybean Machine to impose data dependency orderings conveniently. The mechanism is quite simple. Whenever a calculation is spawned off in parallel, the destination location where the value of the calculation is to be stored is filled with a specially tagged value, called a *context future*, indicating that the value will arrive to the context in the future. When the calculation replies with the value, the future is overwritten with the real value of the computation.

When an access is made to a location in a context, using the value located there, there is the possibility that the value hasn't replied yet. We can tell if the value hasn't returned yet, because it will be filled with a *context future* (c-future) if it hasn't. Any read of a location containing a c-future will cause the processor to fault, (1) saving the processor state in the context object and (2) marking the context as waiting for a c-future. When a reply arrives to a context, the context is checked to see if it is waiting on a c-future. If so, it is queued to be restarted.

| *Advantages* | *Disadvantages* |
|---|---|
| Simple | Large Inertia |
| Transparent | Parallelism Wasted |
| Minimal Synchronization | False Restarts |

Table 8.1: Pros and Cons of Dependency Enforcement by Futures

Let's examine this context-future mechanism in a bit more detail to see what it really provides us and what deficiencies it faces. Table 8.1 itemizes some of the advantages and disadvantages of the future mechanism.

### 8.3.3 Advantages

As we said earlier, the most desirable characteristics of the c-future approach is that it is **simple** to implement and understand. It fits well into the existing system, being "optimistic" — taking advantage of the fault mechanism and the tagged architecture and using contexts.

Being **transparent** to the programmer/compiler writer is desirable as well. No burden is placed on the code generator to explicitly keep track of non-completed tasks. No extra instructions need to be placed in-line to check for the presence of values, or to manipulate semaphores.

Finally, the future approach only pays the price of **synchronisation** *if it is necessary*. If a value returns before it is needed, or if an arm of a conditional is never executed, we will not need to pay the synchronization price[1].

[1]Though we do require all replies to be in before we deallocate a context, so we can re-use context IDs.

### 8.3.4 Disadvantages

On the other hand there are several disadvantages to this approach. The system is subject to **high inertia**. The total cost of halting and saving a context and restarting it when the return value arrives is relatively high. The worst case occurs when we have many dependencies following one after another. Here, we would keep halting and restarting, making very little progress. It can be difficult to gain any momentum, because of the time spent saving and restarting contexts. This case isn't quite so bad if we have other tasks queued up that can take advantage of the free time, and if the replies take a while to arrive (which is likely to be the normal case). The real question is one of balance between computation time and system overhead time.

By controlling execution on the grain size of methods, whenever a sequential execution encounters a c-future value, the entire method will be suspended. Thus once we hit a c-future value, other possibly executable code in the method is not run. This is directly the result of basing the grain of parallelism on the unit of methods, and it has the effect or **wasting parallelism** as opposed to a more fine-grain execution model.

C-futures also can lead to a problem of **false restarts** where a reply for a different slot would restart the context, which would immediately halt on the same c-future again. If we were waiting on variable $A$ to return and a reply to fill variable $B$ arrives, the context would be restarted falsely, and when we read $A$ we will hit the same future and halt again. This is rectified in the prototype implementation, by using the RESOURCE_NEEDED slot of the context to hold the slot number the context need to be filled. When a REPLY arrives, the context is only restarted if it was waiting on the slot the REPLY came to fill.

## 8.4 Distributed Objects

A final system characteristic designed to support efficient high-level language execution is the introduction of distributed objects. A distributed object is one where its state is broken up into segments called *constituent objects*, and scatterred across the processing network. Its purpose is to allow parallel access to different parts of an object.

A single object can only be directly accessed by the node it resides on, and the node it resides on can only run one task, implying that an object can only be computed on by one task at a time. In the absence of coherent caching strategies, this one-object—one-task constraint can potentially severely limit parallelism.

By distributing parts of the object over several nodes we can provide some extra (albeit limited) concurrency. The hope is that this increase of concurrency along with the fact that an object-oriented programming model should provide access to many distinct objects being computed on at once will prevent object bottlenecks from becoming a serious performance hindrance.

The system supports distributed objects by providing (1) allocation and (2) constituent lookup services. When a distributed object is allocated, the system creates constituent objects and scatters them in a reasonable way around the network. Each constituent object has a normal object ID number which is unique for each CO, and a *distributed ID* or DID which is the same for all constituents of a distributed object. This DID contains the information necessary to locate any constituent object.

### 8.4.1 A Distributed ID Format

Figure 8.1 shows a possible format for a distributed ID. The DID knows the number of constituent objects, the hometown node of the first object, and a node-unique serial number. This prototype DID format places a limit of 256 COs per distributed object and 256

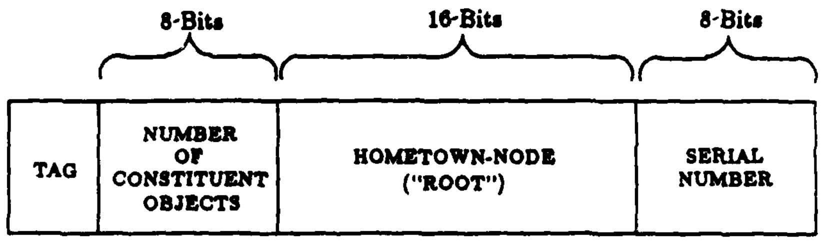


Figure 8.1: Distributed ID Format

distributed objects per node.

### 8.4.2 Dealing out the Constituent Objects

When a distributed object is allocated, we want to have a function that maps each constituent object to a node number. This function should have several properties. It should be (1) easy to compute, it should (2) scatter objects in an acceptable manner.

The goal of distribution is to provide concurrency, so with this aim as the measure of success, any distribution scheme would be equivalent. But, we need to take into account how the processor load is distributed around the network as well. There are two dichotomous goals of constituent distribution, (1) to scatter the objects uniformly across the network so there are no hotspots and (2) to scatter the objects locally to prevent long distance network traffic.

#### Dispersion or Locality?

These seemingly contradictory aims argue against each other. If we scatter objects uniformly, especially if there are very few objects, the data may lie very far away from the majority of the computation. Even though some of the computation will migrate near the data and spawn from there, there still many be a great deal of network traffic caused by

$$\begin{aligned} \text{stride} &= \left\lfloor \frac{\text{nodes}}{\textit{constituents}} \right\rfloor \\ \text{node}_n &= (\text{birthnode} + n \times \textit{stride}) \bmod \text{nodes} \end{aligned}$$

Figure 8.2: Distribution of Constituent Objects

the processes still proceeding from the root of the computation. In time, migration of work may balance the load appropriately, but we still have worries about uniform distribution.

On the other hand, if we clump the constituent objects close together, the computation will cluster around the data, and not hinder the performance of the rest of the network via long distance traffic, but this local hotspot may overwhelm the computational resources of this local area of processors.

## A Simple Dispersal Approach

The first design of the distributed object system leaves this question for further study, and adopts a simple, relatively disperse manner of dealing our constituent objects. We adopt a simple uniform distribution strategy hoping that the load balancing mechanisms incorporated into the system will work effectively. To insure the efficiency of the calculation of the function, we use the simple distribution algorithm shown in figure 8.2. The node numbers we describe are a finite interval of numbers $\{n \in \mathcal{N} : 0 \leq n < \textit{nodes}\}$ we might call *ordinal node numbers* and not the system network address node numbers which encodes the total addressing space of the network. The conversion between the two formats is simple. Figure 8.3 shows some sample distributions for various sized networks, birthnodes, and constituent object counts.

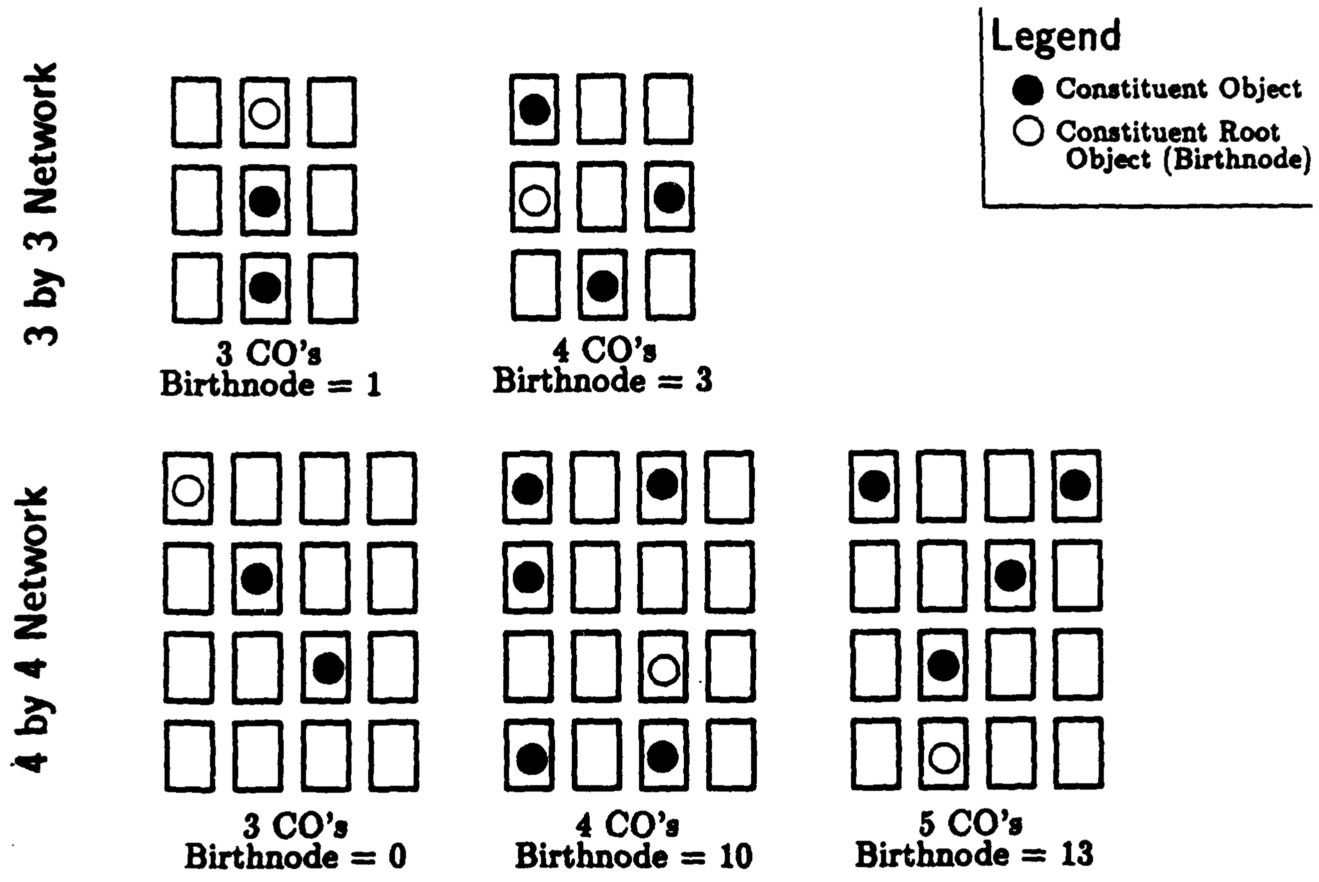


Figure 8.3: Constituent Object Distribution Examples

$$l = \left\lfloor \frac{\text{currentnode}-\text{birthnode}}{\text{stride}} \right\rfloor \times \text{stride} + \text{birthnode}$$

$$r = \left\lfloor \frac{\text{currentnode}-\text{birthnode}+\text{stride}}{\text{stride}} \right\rfloor \times \text{stride} + \text{birthnode}$$

$$\text{if } l < \text{birthnode then } l = l-\text{nodes mod constituents}$$

$$\text{if } r < \text{birthnode then } r = r-\text{nodes mod constituents}$$

$$n = \min(\text{hops}(\text{currentnode},l), \text{hops}(\text{currentnode},r))$$

Figure 8.4: Equations for Choosing a Nearby Constituent Object

## 8.4.3 Choosing a Constituent Object

We now have a first attempt mechanism to assign node numbers to each constituent object. Given a constituent object, we can find the node of its residence. For simplicity, we prevent constituent objects from being migrated. Now, we want to provide an algorithm to choose a constituent object given a DID. We could do this randomly, but in order to take advantage of locality, we want to choose a constituent object that is reasonably close to the current node. We do this by finding the ordinal node numbers of the constituent objects on either side of the current node number ($l$ and $r$ for left and right) and choose the one ($n$) with the minimum distance in x-y hops. We have to be careful about "wraparound". The algorithm is described in figure 8.4.

# Chapter 9

# Issues From a Prototype System

*Keep thy heart with all diligence;*
*for out of it are the issues of life*

— *The Holy Bible, Proverbs 4:23*

This chapter discusses in some detail, relevant issues that occurred in the design and implementation of a prototype operating system. The following topics will be discussed

- The sizing of the BRAT
- How to handle a full translation table
- The scarcity of virtual names
- Out of memory problems
- Queue size
- Queues, stacks, and saving processor state

These situations are troubling enough to require discussion. The actual prototype implementation can be found in an appendix at the end of the thesis. Specifications of the system calls and message handlers can also be found in the appendices.

## 9.1 Sizing the BRAT

To support the global virtual namespace, we use the Birth/Residence Address Table to hold the necessary translation bindings. This serves a purpose similar to a page table in a multi-level paged memory system, or a segment table in a segment addressable memory system. The BRAT needs to hold at least

1. virtual → physical mappings for objects residing on this node
2. virtual → node number links for objects that were born on this node, but now reside elsewhere

### 9.1.1 Memory Limitation

But, due to the small amount of memory on each chip, we face a severe restriction on the number of bindings that can be stored. Reserving room for system data structures, operating system variables, and the heap, we are left with a paltry amount of memory for the BRAT. This will directly limit the amount of objects creatable on a node. We must make a careful compromise between heap size and translation table entries. We must also be able to purge entries from the table when objects are deleted, stressing an efficient storage reclamation strategy.

### 9.1.2 BRAT Use Scenarios

Let's take a look at a few possible scenarios that can occur with object management.

1. There is room left in the heap and the BRAT for more objects to be allocated.
2. There is room left in the BRAT but no more room left in the heap.
3. The heap contains many small objects that don't take up much room, but fill the BRAT, so that no more objects can be created.
4. The heap can be nearly empty, but no more objects can be allocated because the BRAT is full of entries of migrated objects.

The first case is the most desirable one, we wish we could have this happen all the time. The second case is undesirable, but will probably happen reasonably often due to the small memory space. This can be rectified by exporting objects to other nodes to free up heap space. The third and fourth scenarios, however, occur because of lack of translation table space due to the presence of large amounts of resident and/or migrated objects. It is these two cases that we would like to minimize.

The prototype system that was developed assumed 1K of RAM per node. Of this memory, 424 words were reserved for processor and OS data structures. Thus each processor is left with only 600 words to be shared between the heap and the translation table. The question that appears, is how to partition the BRAT and the heap in a reasnable manner.

### 9.1.3 A Prototype Sizing Based On Average Object Size

We have no measures as to object size in our system, but we might be able to suggest a reasonable approximation of, say, 10 words per object[1]. With 2 words of header for each object, this would leave 8 words of object space. So, each object would take up 10 words of heap space and 2 words of BRAT space, allowing $\frac{600}{10} = 60$ objects. But, we also need to reserve room for bindings of objects born on this node, but now residing elsewhere. Let's assume that we pick a limit for this, such as the total number of average-size objects that could fit in the heap. This would allow us to migrate every object and STILL fill the heap with average sized objects. This leaves us with the following equations.

$$\text{heapsize} + \text{bratsize} = \text{freememory}$$

$$\text{residentobjects} = \frac{\text{heapsize}}{10}$$

$$\text{migratedobjects} = \text{residentobjects}$$

$$\text{bratsize} = 2\,(\text{residentobjects} + \text{migratedobjects})$$

[1]Though of course this will depend greatly on the type of program being run.

$$\Longrightarrow \text{heapsize} = \tfrac{5}{7} \times \text{freememory}$$

$$\Longrightarrow \text{bratsize} = \tfrac{2}{7} \times \text{freememory}$$

With 600 words of free space, this leaves the following parameters.

$$\text{heapsize} = 428$$

$$\text{bratsize} = 172$$

In a 4K RAM node, we might expect the following configuration as a reasonable one.

$$\text{heapsize} = 2552$$

$$\text{bratsize} = 1020$$

In the prototype operating system, the BRAT size has been set at 128 words, rather that 172, for ease of implementation.

## 9.2 Running Out of Binding Space

Sooner or later, with even our best efforts at insightful sizing of the BRAT, we will run out of room to make any bindings. There are several conceivable ways of resolving this situation.

1. Throw up your hands and quit.
2. Forward your allocation request to another node.
3. Make the BRAT bigger.
4. "Delegz• " some of the bindings in the BRAT to another node.
5. Change the hometown nodes of some virtual addresses to make other nodes responsible for their bindings.

The current operating system implements choice 1 for the most part. There is also some code to support choice number 2, but this is complicated by the fact that we might not be able to allocate a context (as discussed in an upcoming section). If this mechanism could be made to work, it might be acceptable enough, realizing that any system will break when the nodes begin to run out of memory. The investment in a proper load-balancing policy may alleviate this problem. The operating system also supports the resizing of the BRAT, but because of the hashing mechanism currently used (described in an upcoming section) arbitrary resizing of the BRAT is difficult to do.

The delegation of IDs is possible, but requires some thought. We need a way to specify which IDs are delegated to which nodes, and this should take significanly less storage than would be required to actually store the bindings. We could delegate ranges of IDs to a node, but this node must have room for the range, and when this new node runs out of room, it must also be able to delegate. This is a possibility for the future. The fifth item in the list, changing the birthnodes of virtual addresses would be very expensive requiring some synchronization, and a large broadcast of messages. But, perhaps this could be done during the garbage collection phase, or offline, or at the end of the day as a background job (given a suitably large machine).

## 9.3 Scarcity of IDs

As a related issue, given the virtual ID format of 16 bits of birthnode and 16 bits of serial number, each node can only generate 65536 IDs. In the current system, it is likely that many applications would run through this ID space in a fantastically short amount of time. Of course, the time is dependent on the applications that are run, but we can sketch a rough estimate for how long we can run before running out of IDs on a node.

The following calculations assume a 10MHz processing node where the average instruction length is 1.5 cycles long. We assume that the queue is always full of work to be done. We assume that each message-spawned task work will be 200 instructions long (far above the likely amount). We finally assume that only 10% of the tasks that come in will involve an allocation of an object.

$$\frac{10^7 \text{cycles}}{\text{second}} \times \frac{1 \text{ instruction}}{1.5 \text{ cycles}} \times \frac{1 \text{ task}}{200 \text{ instructions}} \times .1 \frac{\text{allocations}}{\text{task}} = 6667 \frac{\text{allocations}}{\text{second}}$$

At this rate, a node would run out of IDs in 18 seconds. Though these numbers are questionable at best in the absence of actual measurements, it is quite clear that the ID space is compeletely inadequate. We have to have a larger virtual ID, say by having 68 bit words rather than 36 bit words, but in the meantime it might suffice to (1) borrow bits from the node number field or (2) attempting to re-use certain IDs. Borrowing bits would be a short time solution, by limiting our prototype machine to a 1K machine, we could get a 64 fold increase in serial numbers, allowing a node to run for 20 minutes with the assumptions made above. But, for simplicity's sake, the current implementation has not adopted this format. It would be a good idea to do this in the future until we build a machine with larger words.

The second idea is a more interesting research issue. We already reuse context IDs by requiring contexts to have received all replies before they are put on the free list. This way, the amount of IDs reserved for contexts (probably the most frequently allocated object) is significantly cut. There may also be ways of reusing normal object IDs, but a space efficient way of noting these reused IDs may be difficult. Here are a few possible ideas on how to reuse IDs.

1. Keep a fixed size table of free IDs. When an object is freed, the ID will be placed in the table. When an ID is needed, this free table will first be checked. The biggest problem with this approach, is that when the table fills, IDs will not be placed in the table and they will be "lost" forever.
2. Provide a separate routine for allocating "short-lived" objects. These objects would take their IDs from a common, fixed-size pool of consecutive IDs whose freeness could be signified by a single bit for each ID. For example, we might reserve 256 "short-lived" IDs per node. The short-lived IDs' serial numbers might range from 0 to 255 and the pool could be represented by 8 32 bit words signifying an array of 256 bits, where a 0 indicates the ID is in use, and a 1 indicating that it is free. If these objects are truly short-lived, and they represent the bulk of ID requests, then this approach might greatly extend the lifetime by conserving regular IDs.

3. Every now and then, perform an ID "garbage collection and compaction" where all IDs are renamed to consecutive IDs in effect compacting the ID space. This involves similar issues to the mechanism of changing an ID's hometown node number. It seems to be very expensive, but it may be possible to interleave this with the normal garbage collection.

The currently implemented mechanism only reuses context IDs (a fixed amount). No attempt is currently made to reuse other object's IDs.

## 9.4 The Shortage of Memory

Of course, the scarcity of memory per node will also prove to be a problem. The goal is to take advantage of the large collective memory provided by the system (a 4096 node J-Machine with 4K memory per node would have 16 megabytes of primary memory). Load balancing can be used not only in choosing processors to perform work, but also in choosing nodes to allocate memory from. Simple gradient plane approaches [RF87] can be used to cool down memory "hot spots". Garbage collection, expanded memory nodes, and the sweeping of "dusty" objects to offline storage are all possible solutions to the memory shortage problem.

The current prototype operating system kernel takes two approaches to memory. If a message arrives to allocate an object, and there is not enough memory available, the message is forwarded to another node. However, if a process has been running for a while and the node runs out of memory, the calling message cannot simply be forwarded, since some work has already taken place. Instead, the process must have its state saved in a context, and room must be made on this node by evicting certain objects. Unfortunately, there might not be enough memory to allocate a context. A solution out of this trap is to require that there always be one minimal sized context object available for each priority level. A check could be made in the CALL and SEND handlers (and any other message handlers that could fall into these circumstances) for a free context.

## 9.5 Queue Size

Queue sizing also proves to be a problem in the system. Since we want to be able to migrate objects by message sends, an empty queue must always be big enough to hold every object. This means that the queue must be as big as every heap. This is far too costly in terms of memory in the 1K node prototype, and we have not attempted to make a fix. It would always be possible, though admittedly tedious, to send messages in "chunks" that would be able to fit in the queues.

## 9.6 Suspension and Processor State

Whenever a process suspends and plan on restarting later, it must be able to save its processor state. This normally means its register set, but we must not forget about two other forms of processor state, queues and stacks. When we suspend and there is a message we want to save in the queue, we copy it out into a heap object and set the message pointer to point to the object instead of the queue. Stacks are more of a difficulty to save and restore, and we have decided to explicitly prohibit the saving of stack frames. So, the operating system is given the task of insuring it will never have to suspend and restart with information on the stacks. This was a source of much personal misery during the implementation of the OS (though certainly less than there would have been without the existance of stacks).

## 9.7 Summary

This chapter has touched on just a few of the difficulties in the design of the Jellybean Operating System Software. Some are due to inadequacies in hardware or scale, some are due to lack of behavioral measurements, and some due to lack of insight. These will most likely become thoroughly examined as the machine design progresses into subsequent stages.

# Chapter 10

# Performance Evaluation

*Never promise more than you can perform.*

— *"Publilius Syrus", Maxim 528*

This chapter provides a quantitative performance evaluation of several important system services. Though the prototype implementation is certainly not optimal in any way, it should be a reasonable approximation of an actual working operating system kernel, and as such, the numbers presented in the chapter should be useful for the design and tuning of the rest of the Jellybean system. In addition, we should be able to see what parts of the system need fixing, before the machine is fabricated.

## 10.1 The Virtual Binding Tables

The virtual name manager is composed of five system routines nested in the hierarchy shown in figure 10.1. The BRAT itself is composed of a 128 word binding table of 64 2-word bindings. Words are entered by a *linear probing* [Sed83] scheme where a hash function determines the first choice for the location of the binding, and a linear search is performed

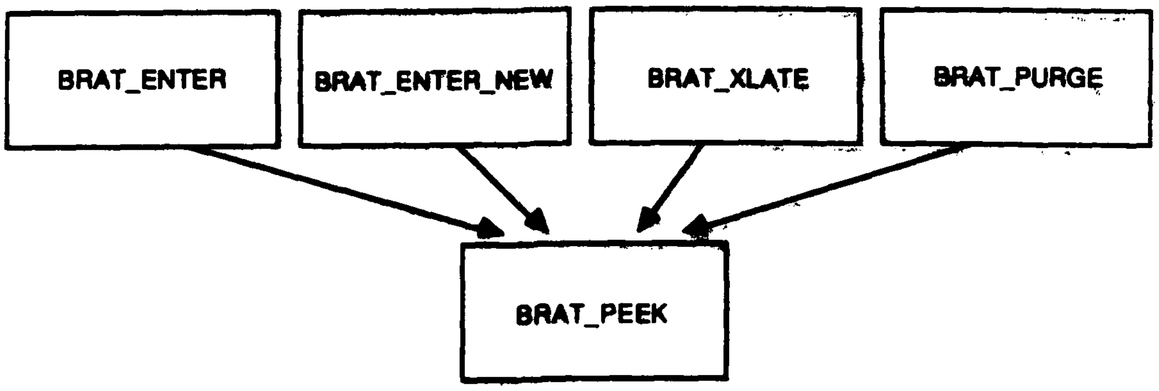


Figure 10.1: The Hierarchy of the Virtual Name Manager

from there. This linear search can take a significant amount of time (at least on the scale of average task size), so we need (1) an efficient algorithm and (2) a successful hashing scheme. The remainder of this section examines the execution time of each BRAT routine and presents some very preliminary hashing measurements.

### 10.1.1 Instruction Counts

The BRAT_PEEK system call is the core to all of the virtual name services. It takes a key to hash and a data word to match (not necessarily the same, since you might want to look for the first NIL slot where a certain key could be placed, as is done when adding new entries). The key is hashed, providing the index into the table, and a linear search with wraparound proceeds from here. The cost of this call is between 22 and 540 instructions, based on how far the search has to progress. A reasonable cost approximation, $C_{\text{peek}}$, for a search that finds the data in the $n^{\text{th}}$ slot is $22 + 8 \times (n - 1)$ steps.

The rest of the BRAT calls utilize this BRAT_PEEK routine.

- BRAT_XLATE looks up a binding in the BRAT and takes $27 + C_{\text{peek}}$ steps to complete.
- BRAT_PURGE searches the BRAT until it finds the first binding of the specified word, and removes it from the table. This takes $30 + C_{\text{peek}}$ steps to complete.
- BRAT_ENTER_NEW adds a new entry to the BRAT without first removing any previous bindings. It accomplishes its task in $32 + C_{\text{peek}}$ steps.
- The most expensive routine, potentially, is the BRAT_ENTER routine. This is like BRAT_ENTER_NEW, but it first removes a previous binding, requiring another BRAT search. This can take as much as $32 + 2 \times C_{\text{peek}}$ steps.

### 10.1.2 Effectiveness of Linear Probing

Evidently, the crucial factor in the effectiveness of the BRAT routines is the cost of peeking through the BRAT, $C_{\text{peek}}$, which is a linear function of how far away from the expected hash spot the value resides. What the average distance in hash steps will be for a typical machine, depends greatly on (1) the application that is being run, (2) how storage reclamation is handled, (3) and what is done when the BRAT overflows — all issues needing further study. Nonetheless, I would like to proceed with an informal, ad hoc analysis, based on reasonable estimates and educated guesswork. The rationale is to see if the linear probing strategy seems to generally work — by that, meaning that the average number of steps is small until the entry is found[1].

[1]It is not obvious that this will so. In fact, it is quite easy to be concerned that this linear rehashing approach might actually work itself into a steady state where entries were always very far away from where they were supposed to be.

The following data was generated by a simulation program called *bratsim* that takes an input pattern of references and simulates their effect on the BRAT. The size and maximum fullness of the BRAT is specifiable. The simulator takes each reference and looks it up in the BRAT.

- If the reference is in the BRAT, it records the number of steps away from where it should be.
- If the reference is not in the BRAT, it is entered as soon as possible after its hashed spot.
- When names get entered, some may be arbitrarily deleted to maintain a maximum full percentage.
- If the BRAT fills, a random slot will be emptied.

The reference pattern generator is also based on initial approximations, generating patterns possibly likely in applications we envision running. It is currently configured with the following parameters: 10% new IDs, 20% context IDs, 35% recent IDs to simulate locality, 20% less local IDs, and 15% very random IDs to simulate class/selector bindings, method IDs and other references following less of a pattern. I would expect this estimate to be conservative.

Based on these estimates, and the reclamation model presented above, we can chart how many steps away from the hashed slot particular IDs land when they are entered. For a 64 word table, this is graphed in figure 10.2. We see an asymptotic function relating BRAT space used and the locality of entries to their intended slots. For the 64 row example, the system begins to be unmanageable after the BRAT becomes more than 60 – 70% full.

Figure 10.3 shows the effect of doubling the BRAT size. The trend is still rapidly increasing, but the gains we get in terms of object storage may outweigh the extra steps involved in lookup. The flatness of the middle portion, from 40 – 60% hints at a desirable operating region.

So, now I would like to suggest educated guesses to the answers to the following two questions.

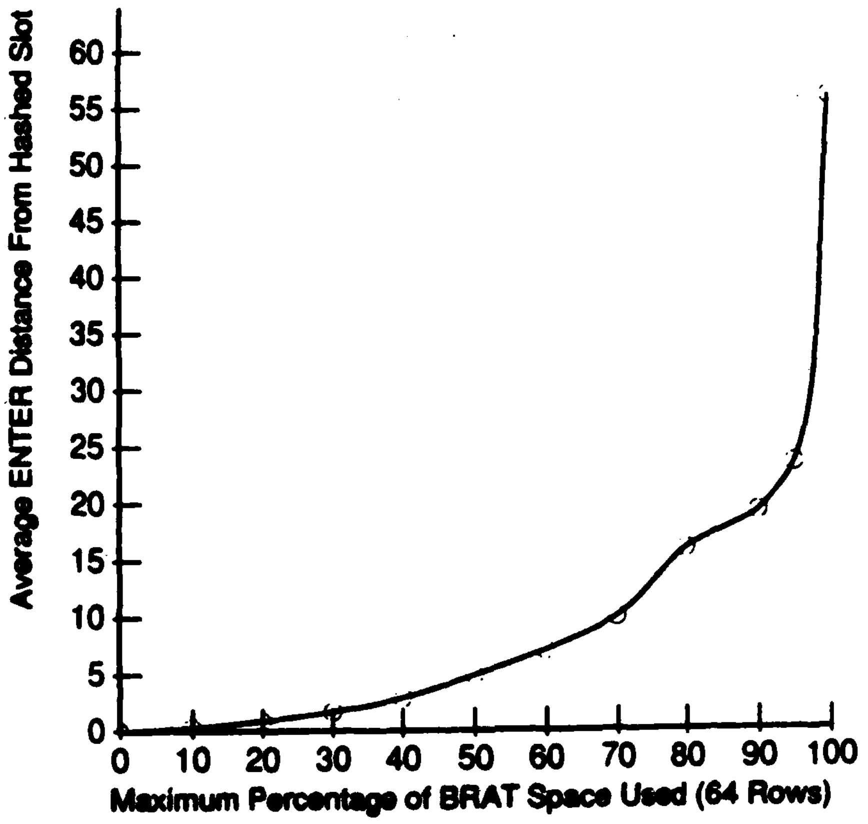


**Figure 10.2: 64 Row BRAT Enter Distances from Hashed Slot**

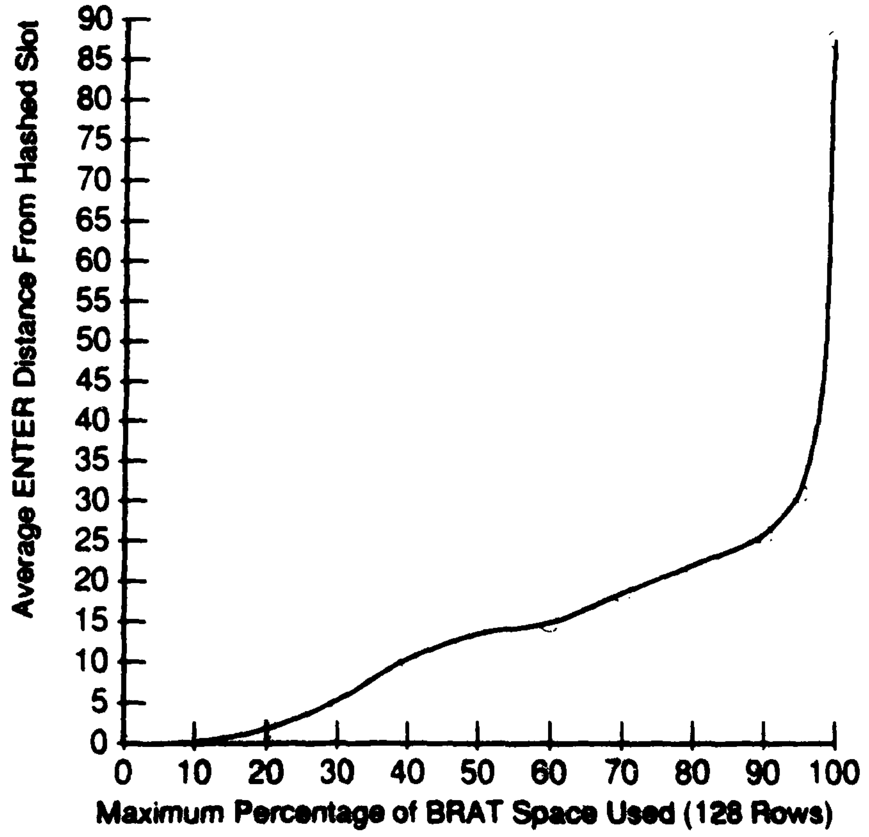


Figure 10.3: 128 Row BRAT Enter Distances from Hashed Slot

1. How full should we allow the BRAT to get?

2. How large should the BRAT be?

In the last few paragraphs, I indicated the severity of the BRAT filling problem. After 70% capacity, the BRAT's performance becomes intolerable. For this reason, I suggest that 70% capacity should be an absolute maximum for BRAT size, and the normal operating size should not usually exceed 50%. I propose this as the answer for question 1.

Question number 2 can be answered by adapting the analysis presented in the last chapter. The new constraint equations become.

$$\text{heapsize} + \text{totalbratsize} = \text{freememory}$$
$$\text{residentobjects} = \frac{\text{heapsize}}{10}$$
$$\text{migratedobjects} = \text{residentobjects}$$
$$\text{bratspaceused} = 2\,(\text{residentobjects} + \text{migratedobjects})$$
$$\text{bratspaceused} = .7 \times \text{totalbratsize}$$
$$\Longrightarrow \text{totalbratsize} = \tfrac{4}{11} \times \text{freememory}$$
$$\Longrightarrow \text{heapsize} = \tfrac{7}{11} \times \text{freememory}$$

With 600 words of free space, this reserves 218 words for the BRAT and 382 words for the heap. This will hopefully be a more accurate value, though it is not a power of 2, which will complicate the hashing slightly.

The efficient manipulation of the BRAT is crucial to the success of the Jellybean system. Future study is needed to evaluate hashing functions, and perhaps a form of *linear re-hashing* is desired, where the first hash is followed by a subsequent number of other hashes instead of a linear search. In addition, once real applications are run, we can get a better idea how the system will behave. Likewise, the translation buffer performance needs analysis, as this will indicate how often BRAT lookup occurs.

## 10.2 Object Allocation

A common task of the Jellybean Operating System Software is to allocate objects from the heap. This section will examine how costly this operation can be.

Figure 10.4 describes the nesting of services required to perform the NEW system call. The ALLOC routine takes 24 instructions, it takes 19 instructions to generate a new ID and it takes 32 + $C_{\text{peek}}$ instructions to enter a new ID into the BRAT. With 20 cycles for inter-module glue, the NEW system call takes 95 + $C_{\text{peek}}$ instructions. According to the BRAT analysis results, if we operate at less than 70% full, we will have to take less than 10 steps to enter a new ID, this would indicate that $C_{\text{peek}} = 94$ steps and therefore, NEW should take 95 + 94 = 189 instructions. At best, with 0 steps to search, the NEW call would take 117 steps.

## 10.3 Context Allocation

Another commonly executed routine is the NEW_CONTEXT system call. As described in chapter 5, this service was expected to be expensive enough to merit special treatment. The context free list was developed to provide a pool of pre-allocated contexts for fast context allocation. The flowchart in figure 10.5 shows the steps taken by routine. Note that if the requested context is of an abnormal size, or if there are no pre-allocated contexts on the free list, the NEW routine is called to allocate a new object. Requesting an abnormally sized context takes $25 + C_{\text{new}}$ instructions, allocating a context when node are on the free list takes $27 + C_{\text{new}}$ instructions, but allocating a context off the free list takes only 20. If we can keep contexts in the pool, we will do well.

Freeing contexts is also fast, taking only 25 instructions. This is only about 10% of the time it used to take to perform this operation, when we were required to purge the

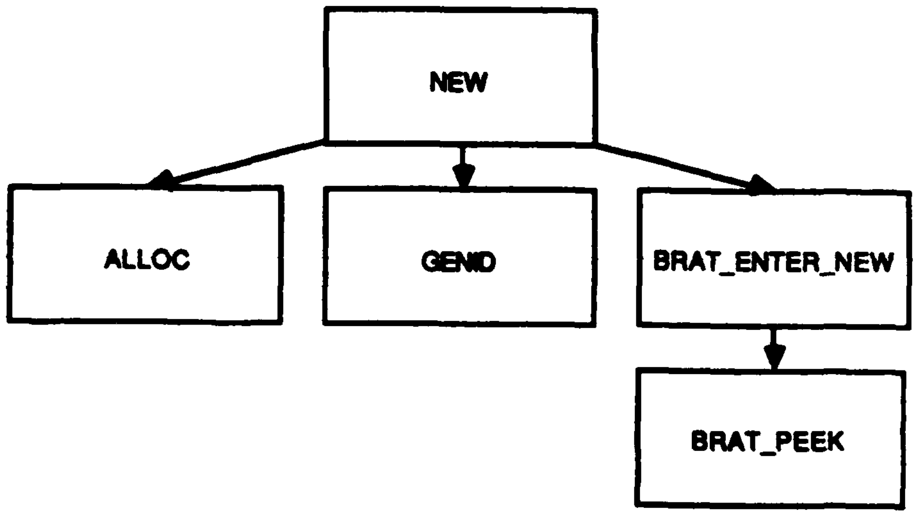


**Figure 10.4: Nesting of Services for the NEW System Call**

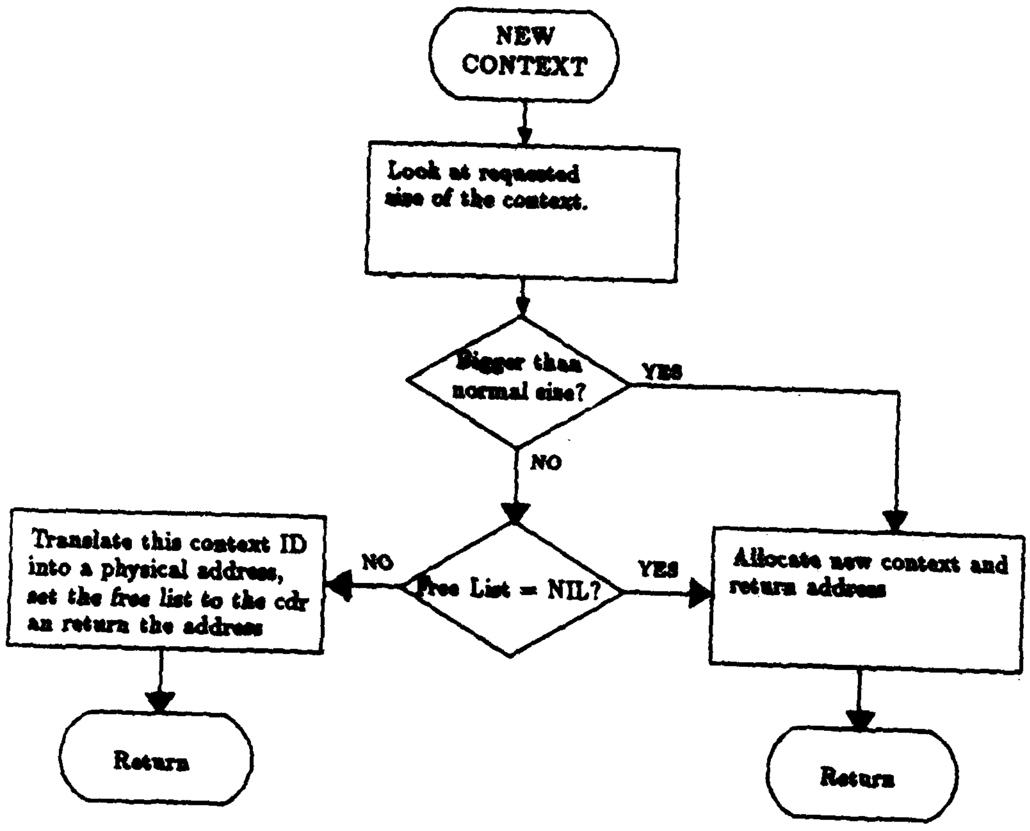

NEW CONTEXT
Look at requested size of the context.
Bigger than normal size?
YES
NO
Translate this context ID into a physical address, set the free list to the cdr an return the address
NO
Free List = NIL?
YES
Allocate new context and return address
Return
Return

Figure 10.5: Flowchart for the NEW_CONTEXT System Call

old context ID, generate a new one, and place the new ID in the context and BRAT. By preventing late replies to contexts, we have prevented this performance loss.

## 10.4 Boot Code and Message Handlers

Let's conclude the chapter with a brief discussion of the complexity of the Bootstrap code and several message handlers. The boot code is run when each processor is powered up, and places the processor in a runnable state. All together, it takes 5005 steps to boot the processor. This is made up of 4103 steps to erase the memory, 481 steps to initialize the context free list with 3 contexts, 247 steps to fill the exception vector table, 86 steps to fill the extended call table and 72 steps to set up the stacks, queues and other values.

The WRITE message handler takes $8 + 7 \times l + 3$ steps to send $l$ words of data. The READ message handler takes 8 steps to read an empty message, or $7 + 5 \times (l - 1)$ steps to read a block of data of length $l$.

The CALL message handler can exhibit several possible times. If the method being CALLed is local, it only takes 6 instructions to start it executing. If the method is local, but not in the cache, it takes $64 + C_{\text{peek}}$ steps, because the XLATE exception handler takes $58 + C_{\text{peek}}$ steps to complete. If the method is not local, message sends are involved making it more difficult to analyze.

## 10.5 ROM Size

Out of the 1024 words reserved for ROM, the operating system prototype uses 760.

## 10.6 Summary

This section presented a brief performance evaluation of several important parts of the Jellybean system. In addition to analyzing the cost of routines, several more fundamental issues were noticed. These are itemized below.

- The BRAT needs to be searched efficiently. The *linear probing* method used can take a significantly long time if values get placed far from their intended position.
- Based on preliminary simulation, the performance becomes unacceptable when the BRAT gets to 60 to 70 percent full. We can choose a maximum fullness, and derive the BRAT and heap sizes based on the fullness value and the expected size of objects.
- We note that even with an insightful configuration of the BRAT, a translation cache is required. The configuration of the cache is left to further study.
- Creating a new object is more expensive than we would like (a minimum of 117 instructions). This could be optimized with clever coding, but not much more performance could be gained by this manner. The problem is more fundamental resting on the performance of the cache and the BRAT lookup.
- The caching of free contexts seems to work well. Creating a new context requires only 20 instructions if there is a context on the free list (and assuming we don't get a translation fault). This is compared to a minimum of 144 instructions without a context on the free list. Freeing a context is also fast, only 25 instructions.
- Calling a local method takes only 6 instructions if the method is local and its translation is in the cache! If it is not in the cache, performance again suffers, requiring a minimum of 86 instructions.

Table 10.1 summarizes some of the more important performance statistics presented in this chapter.

| *Routine* | *Instruction Count* | *Notes* |
|---|---|---|
| BRAT_PEEK | $C_{\text{peek}} = 22 + 8 \times (n - 1)$ | n = slots to search |
| BRAT_XLATE | $27 + C_{\text{peek}}$ | |
| BRAT_PURGE | $30 + C_{\text{peek}}$ | |
| BRAT_ENTER_NEW | $32 + C_{\text{peek}}$ | |
| BRAT_ENTER | $32 + 2 \times C_{\text{peek}}$ | maximum |
| ALLOC | 24 | |
| GENID | 19 | |
| NEW | $95 + C_{\text{peek}}$ | |
| NEW_CONTEXT | 20 | with context on free list |
| | $27 + C_{\text{peek}}$ | no context on free list |
| FREE_CONTEXT | 25 | |
| CALL_MSG | 6 | with method ID in cache |
| | $64 + C_{\text{peek}}$ | method ID not in cache |

Table 10.1: Timings for Common System Services

# Chapter 11

# Conclusions

*All's well that ends well*

— SHAKESPEARE, in *All's Well That Ends Well IV*

*There is a time for many words,*
*and there is also a time for sleep.*

— HOMER, in *The Iliad, XI*

## 11.1 Summary

The Jellybean Operating System Software is a prototype operating system kernel for the Jellybean Machine. Its duties include object-based storage allocation, virtual distributed naming, object migration, process definition and control, local and remote process execution, and the support of an object-orient calling model.

This thesis described the JOSS in some detail, its successes and weaknesses. The report also talks about issues in the future Jellybean operating system that were not implemented in the prototype because of lack of support, study and time. These include storage reclamation, resource distribution bureacracies, and distributed objects. These will most likely become important parts of the Jellybean operating environment in the future.

Several deficiencies may exist in the current system. Performance-wise, searching the translation table may well be too slow. Several solutions can be proposed including (1), increasing the size of the BRAT and decreasing the fullness, (2) experimenting with various hashing functions and (3) providing an effective translation buffer. Memory shortages may provided a significant problem, and this will place an extra burden on reclamation attempts, which are already made difficult because of the problem of *travelling references*.

On the other hand, if the cache works well, and if the BRAT is not very full, the whole system seems to perform admirally. Method invocations are powerful but fast. The context free list allows rapid creation and reuse of contexts. The global naming system and migration provides a high degree of flexibility.

## 11.2 Suggestions for Further Study

This thesis scratched the surface of many interesting research issues, many of which I for one would be eager to investigate.

In the area of performance evaluation, the configuration and simulation the translation buffer and BRAT in a real life environment is important to the success of the Jellybean Machine. Also of practical as well as theoretical interest would be the study and evaluation of distribution hierarchies and the various manifestations of how to handle virtual hints.

Reclamation is an important potential area of research. An efficient mechanism to collect garbage over a distributed network would be of general interest as well, especially if some incremental form of collection can be developed. Policies for handling out-of-memory conditions on processing nodes is also attractive, involving selective migration of objects.

Finally, load and resource balancing policies need to be investigated, especially since each processor can quickly become overwhelmed (being limited in power and memory capacity). Simple gradient plane approaches might be attempted where load spreads to where it is lower. Network analysis will also be an important factor.

## 11.3 Hopes

The Jellybean Machine has the potential of being an important step in the development of multicomputer networks. It is my hope that further study will be encouraged so that the difficulties of machines of this genre can be resolved (memory shortages, expensive name translation, no caching of mutable objects, need for resource balancing, etc.) and they can show their benefits as scalable, programmable processors.

# Appendix A

# Operating System Equates

```
; ************************************************************************
; ***                                                                  ***
; ***                                                                  ***
; ***     Designed and implemented  by the members of the Concurrent   ***
; ***     VLSI Architecture Group at the Massachusetts Institute of    ***
; ***     Technology.                                                  ***
; ***                                                                  ***
; ***   Copyright (C) 1986, 1987  Massachusetts Institute of Technology ***
; ***                         ALL RIGHTS RESERVED                      ***
; ***                                                                  ***
; ***       No copy of this source code may be made by any means,      ***
; ***       electronic or otherwise,  without prior permission of      ***
; ***       the Massachusetts Institute of Technology.                 ***
; ***                                                                  ***
; ************************************************************************

;-----------------------------------------------------------------
; OS.MDP
;
; This file contains operating system labels & stuff
;-----------------------------------------------------------------

;       ********************
;       Useful system values
;       ********************

LABEL   SYS_LEN_BITS                    =       10
LABEL   SYS_LEN_MASK                    =       %1111111111
LABEL   SYS_ID_NODE_BITS                =       16
LABEL   SYS_ID_ID_BITS                  =       16
LABEL   SYS_ID_ID_MASK                  =       %1111111111111111
LABEL   SYS_ID_NODE_MASK                =       %1111111111111111
LABEL   SYS_CLASS_MASK                  =       %1111111111111111
LABEL   SYS_CLASS_BITS                  =       16
LABEL   SYS_SELECTOR_MASK               =       %1111111111111111
LABEL   SYS_SELECTOR_BITS               =       16
LABEL   SYS_OP0_BITS                    =       7
LABEL   SYS_OP1_BITS                    =       2
LABEL   SYS_OP2_BITS                    =       2
LABEL   SYS_OP0_MASK                    =       %1111111
LABEL   SYS_UNCHECKED                   =       (1<<31)
LABEL   SYS_UNC                         =       SYS_UNCHECKED
LABEL   SYS_ABSHADOW                    =       (1<<8)
LABEL   SYS_ABS                         =       SYS_ABSHADOW
LABEL   SYS_INVADR                      =       (1<<30)
LABEL   SYS_MARK_MASK                   =       (1<<31)
LABEL   SYS_COPY_MASK                   =       (1<<30)
LABEL   SYS_REL_MASK                    =       (1<<31)
LABEL   SYS_UNMOVABLE_MASK              =       (1<<29)

;       ***********
;       XLATE Modes
;       ***********

LABEL   XLATE_OBJ                       =       0
LABEL   XLATE_ID_TO_NODE                =       1
LABEL   XLATE_METHOD                    =       2
LABEL   XLATE_LOCAL                     =       3

;       *******************
;       Temporary locations
;       *******************

LABEL   TEMP0                           =       0
LABEL   TEMP1                           =       1
LABEL   TEMP2                           =       2
LABEL   TEMP3                           =       3
LABEL   TEMP4                           =       4
LABEL   TEMP5                           =       5
LABEL   TEMP6                           =       6

;       **********
;       Memory Map
;       **********

LABEL   OS_P0_TEMPS_BASE                =       0
LABEL   OS_P0_TEMPS_LENGTH              =       8
LABEL   OS_P1_TEMPS_BASE                =       8
LABEL   OS_P1_TEMPS_LENGTH              =       8
LABEL   OS_EVECTORS_BASE                =       16
LABEL   OS_EVECTORS_LENG...             =       48
LABEL   OS_P0_STACK_BASE                =       64
LABEL   OS_P0_STACK_LENGTH              =       32
LABEL   OS_P1_STACK_BASE                =       96
LABEL   OS_P1_STACK_LENGTH              =       32
LABEL   OS_QUEUE0_BASE                  =       128
LABEL   OS_QUEUE0_MASK                  =       127
LABEL   OS_CACHE_BASE                   =       256
LABEL   OS_CACHE_MASK                   =       63
LABEL   OS_QUEUE1_BASE                  =       320
LABEL   OS_QUEUE1_MASK                  =       31
LABEL   OS_VARS_BASE                    =       352
LABEL   OS_VARS_LENGTH                  =       16
LABEL   OS_MCACHE_BASE                  =       376
LABEL   OS_MCACHE_LENGTH                =       32
LABEL   OS_XVECTORS_BASE                =       408
LABEL   OS_XVECTORS_LENGTH              =       16
LABEL   OS_LOCKED_BASE                  =       424
LABEL   OS_LOCKED_LENGTH                =       0
LABEL   OS_INITIAL_BRAT_LENGTH          =       128
LABEL   OS_INITIAL_BRAT_MASK            =       (OS_INITIAL_BRAT_LENGTH-1)&(~1)

;       **************************
;       Locations of OS Variables
;       **************************

LABEL   VAR_FREETOP                     =       OS_VARS_BASE + 0
LABEL   VAR_BRAT_BASE                   =       OS_VARS_BASE + 1
LABEL   VAR_BRAT_LENGTH                 =       OS_VARS_BASE + 2
LABEL   VAR_BRAT_HASH_MASK              =       OS_VARS_BASE + 3
LABEL   VAR_ROM_START                   =       OS_VARS_BASE + 4
LABEL   VAR_NEXT_ID                     =       OS_VARS_BASE + 5
LABEL   VAR_LAST_ID                     =       OS_VARS_BASE + 6
LABEL   VAR_MCACHE_BASE                 =       OS_VARS_BASE + 7
LABEL   VAR_MCACHE_LENGTH               =       OS_VARS_BASE + 8
LABEL   VAR_MCACHE_OVERFLOW_LIST        =       OS_VARS_BASE + 9
LABEL   VAR_CFREE_LIST                  =       OS_VARS_BASE + 10
LABEL   VAR_HEAP_BASE                   =       OS_VARS_BASE + 11
LABEL   VAR_NET_WIDTH                   =       OS_VARS_BASE + 12
LABEL   VAR_NET_HEIGHT                  =       OS_VARS_BASE + 13

;       **********
;       Tag Values
;       **********

LABEL   TAG_SYM                         =       0
LABEL   TAG_INT                         =       1
LABEL   TAG_BOOL                        =       2
LABEL   TAG_ADDR                        =       3
LABEL   TAG_IP                          =       4
LABEL   TAG_MSG                         =       5
LABEL   TAG_A                           =       6
LABEL   TAG_B                           =       7
LABEL   TAG_C                           =       8
LABEL   TAG_D                           =       9
LABEL   TAG_E                           =       10
LABEL   TAG_F                           =       11
LABEL   TAG_CS                          =       TAG_D
LABEL   TAG_OBJHEAD                     =       TAG_E
LABEL   TAG_OBJID                       =       TAG_F
LABEL   TAG_INST0                       =       12
LABEL   TAG_INST1                       =       13
LABEL   TAG_INST2                       =       14
LABEL   TAG_INST3                       =       15

;       **************************
;       Exception Vector Locations
;       **************************

LABEL   EVECTORBASE                     =       OS_EVECTORS_BASE

LABEL   FAULT_BKGD                      =       EVECTORBASE
LABEL   FAULT_DBLFAULT                  =       EVECTORBASE + 1
LABEL   FAULT_ILGINST                   =       EVECTORBASE + 2
LABEL   FAULT_ILGADRMD                  =       EVECTORBASE + 3
LABEL   FAULT_ACCESS                    =       EVECTORBASE + 4
LABEL   FAULT_EARLY                     =       EVECTORBASE + 5
LABEL   FAULT_LIMIT                     =       EVECTORBASE + 6
LABEL   FAULT_INVADR                    =       EVECTORBASE + 7
LABEL   FAULT_MSG                       =       EVECTORBASE + 8
LABEL   FAULT_QUEUE                     =       EVECTORBASE + 9
LABEL   FAULT_SEND                      =       EVECTORBASE + 10
LABEL   FAULT_XLATE                     =       EVECTORBASE + 11
LABEL   FAULT_RANGE                     =       EVECTORBASE + 12
LABEL   FAULT_PUSH                      =       EVECTORBASE + 13
LABEL   FAULT_POP                       =       EVECTORBASE + 14
LABEL   FAULT_OVERFLOW                  =       EVECTORBASE + 15
LABEL   FAULT_TYPE                      =       EVECTORBASE + 17
LABEL   FAULT_IA                        =       EVECTORBASE + 18
LABEL   FAULT_IB                        =       EVECTORBASE + 19
LABEL   FAULT_IC                        =       EVECTORBASE + 20
LABEL   FAULT_ID                        =       EVECTORBASE + 21
LABEL   FAULT_IE                        =       EVECTORBASE + 22
LABEL   FAULT_IF                        =       EVECTORBASE + 23

;       *******
;       Classes
;       *******

LABEL   CLASS_CONTEXT                   =       1
LABEL   CLASS_METHOD                        =    2
LABEL   CLASS_MESSAGE                       =    3
LABEL   CLASS_INT                           =    512

;       ********************
;       System Call Values
;       ********************

LABEL   TRAP_NEW_CONTEXT                    =    0
LABEL   TRAP_FREE_CONTEXT                   =    1
LABEL   TRAP_XFER_ID                        =    2
LABEL   TRAP_XFER_ADDR                      =    3
LABEL   TRAP_ID_TO_NODE                     =    4
LABEL   TRAP_NEW                            =    5
LABEL   TRAP_MALLOC                         =    6
LABEL   TRAP_GENID                          =    7
LABEL   TRAP_VERSION                        =    8
LABEL   TRAP_BRAT_PEEK                      =    9
LABEL   TRAP_SWEEP                          =    10
LABEL   TRAP_FREE_SPECIFIED_CONTEXT         =    11

LABEL   TRAP_XCALL                          =    14
LABEL   TRAP_DIE                            =    15

;       **********************
;       Extended Call Values
;       **********************

LABEL   XCALL_BRAT_ENTER                    =    1
LABEL   XCALL_BRAT_XLATE                    =    2
LABEL   XCALL_BRAT_PURGE                    =    3
LABEL   XCALL_MIGRATE_OBJECT                =    4
LABEL   XCALL_BRAT_ENTER_NEW                =    5

;       *********************
;       Object Field Offsets
;       *********************

LABEL   OBJECT_HDR                          =    0
LABEL   OBJECT_ID                           =    1

LABEL   CONT_PSTATE_OFFSET                  =    2
LABEL   CONT_NEXT_CONTEXT                   =    3
LABEL   CONT_RESOURCE                       =    4

LABEL   CONT_NORMAL_SIZE                    =    13

LABEL   PSTATE_ID0                          =    0
LABEL   PSTATE_ID1                          =    1
LABEL   PSTATE_ID2                          =    2
LABEL   PSTATE_ID3                          =    3
LABEL   PSTATE_R0                           =    4
LABEL   PSTATE_R1                           =    5
LABEL   PSTATE_R2                           =    6
LABEL   PSTATE_R3                           =    7
LABEL   PSTATE_IP                           =    8

LABEL   CONT_PSTATE_SIZE                    =    9

;       ***********
;       Handler IDs
;       ***********

LABEL   HANDLER_INSTALL_METHOD              =    TAG_OBJID:0
LABEL   HANDLER_LOOKUP_METHOD               =    TAG_OBJID:1
```

# Appendix B

# Operating System ROM Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;                                                                  ;;;;
;;;;                                                                  ;;;;
;;;;     Designed and implemented  by the members of the Concurrent   ;;;;
;;;;     VLSI Architecture Group at the Massachusetts Institute of    ;;;;
;;;;     Technology.                                                  ;;;;
;;;;                                                                  ;;;;
;;;;   Copyright (C) 1986, 1987  Massachusetts Institute of Technology ;;;;
;;;;                          ALL RIGHTS RESERVED                     ;;;;
;;;;                                                                  ;;;;
;;;;       No copy of this source code may be made by any means,      ;;;;
;;;;       electronic or otherwise,  without prior permission of      ;;;;
;;;;       the Massachusetts Institute of Technology.                 ;;;;
;;;;                                                                  ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;---------------------------------------------------------------------
;
;       ROM.MDP
;
;       This file contains system kernel routines for the MDP ROM
;
; Edit History (started 6/23/87)
;
;       Who     Date            What
;       ---     ----            ----
;
;       Brt     6/23/87         Added STAT_x labels.  Added ROM_SIZE
;                               calculations.  Changed temporary use to
;                               avoid bashing in conjunction with dependency
;                               graph, and larger temporary space.  Fault
;                               handlers now use FTEMPs instead of TEMPs.
;                               New trashing specification to make variable
;                               use clearer.
;       Brt     6/24/87         More work on method mode of XLATE_EXC.
;       Brt     6/26/87         Stack testing code & boot initialization.
;                               Started converting trap routines from TEMP
;                               to stack conventions.
;       Brt     6/29/87         Continued converting to stack conventions
;       Brt     6/30/87         Removed stack conventions
;       Brt     7/06/87         Inserted stack conventions
;       Brt     7/08/87         Started conversions to V8, including the
;                               new register instructions
;       Brt     7/10/87         Continued conversions
;       Brt     7/13/87         Put some initial garbage collection attempts in
;       Brt     7/17/87         Put in BRAT manipulation traps.  We need more
;                               trap vectors for system calls.  So, add a
;                               system call location to use another table
;                               sometime.
;       Brt     7/28/87         Switched to version 8.
;       Brt     8/05/87         Upgraded XLATE_EXC
;       Brt     8/10/87         Finished code for XLATE_EXC & method caching,
;                               but haven't tested it yet.  Fixed some bugs
;                               in the BRAT manipulators.
;       Brt     8/11/87         Tested XLATE_EXC & method caching code.
;                               There is a bug after the METHOD_REQUEST_REPLY
;                               that causes a MSG fault.  I think that the
;                               METHOD_REQUEST_REPLY message has a length that
;                               is maybe 1 too small, so when the
;                               RESTART_CONTEXT message arrives, the last
;                               word of the previous message is used as the
;                               message header???  Also updated os.mdp file.
;       Brt     8/12/87         Fixed the method caching length-of-message
;                               problem.  Made XFER restore data registers
;                               and ID registers, and not try to reXLATE
;                               A0 if it's ID register is nil.
;       Brt     2/06/88         Modified context format to move processor
;                               state to the end.  Updated OS.MDP.
;       Brt     2/10/88         Added FREE_CONTEXT_TRP & FREE_CONTEXT_MSG.
;                               Fixed OS.MDP that has OS vars in wrong place
;       Brt     2/16/88         Added NEW_METHOD_MSG, ID_TO_NODE_TRP,
;                               placed local XLATE in XLATE_EXC (for ID_TO_NODE
;                               as well as other simple uses of XLATE), wrote
;                               SEND_MSG.
;       Brt     2/19/88         Made XFER free contents.  Fixed up SWEEP_TRP.
;       Brt     2/22/88         Finished & tested heap compactor.
;       Brt     3/04/88         Changed ID_TO_NODE_TRP, Removed XLATE_NOVR
;                               mode - replaced with XLATE_LOCAL and in-line
;                               code within SEND_MSG.  Added XLATE_ID_TO_NODE
;                               mode to XLATE.
;       Brt     -'08/88         Added locked down region to memory map.  Made
;                               LOCKHEAP equivalent to PUSH I,MOVE TRUE,I and
;                               UNLOCKHEAP to POP I.
;       Brt     3/16/88         Added method cache overflow list support.
;                               Added extended system call mechanism.
;       Brt     3/18/88         Added "copy" bit to method headers.  Cached
;                               methods are now distinguished by this copy
;                               bit rather than using the method directory
;                               also for this purpose.  Started INVADR_EXC
;                               handler.
;       Brl     5/03/88         Last minute change to improve performance
;                               of allocating objects.  Made new xcall,
;                               XCALL_BRAT_ENTER_NEW that enters a new ID that
;                               is guaranteed not to be in there already.
;                               Removed code to generate new ID from
;                               FREE_CONTEXT system call.
;
;----------------------------------------------------------------------

;
; Note: Both Main code & exception code use some TEMPs.  Be sure
;       that they don't bash each other!!!!!
;

        ASM "os.mdp"                            ; OS include file

        MODULE
        ORG     1024

;****************************************************************************
;
;                       B O O T   C O D E
;
;****************************************************************************
```

```
;------------------------------------------------------------------------
; BOOT -- This routine contains the cold boot MDP code
;

BOOT:
                ; Find how much RAM we have

        DC      1024                            ; This is a hack to fill
                                                ;  R0 with the amount of RAM

                ; Clear memory

        MOVE    R0,R1                           ; Copy amount of RAM to R1
        MOVE    R0,R2                           ; Also copy to R2
        MOVE    NIL,R0
_BOOT_CLR:
        BZ      R1,^_BOOT_CLRDONE               ; If loop done, break out
        SUB     R1,1,R1                         ; Decrement R1
        MOVE    R0,[R1,A0]                      ; Stick NIL to address
        BR      ^_BOOT_CLR                      ; Loop
_BOOT_CLRDONE:

                ; Save the RAM size in the OS variable, now that RAM is clear

        DC      VAR_ROM_START                   ; R0 <- Offset to ROM_START var
        MOVE    R2,[R0,A0]                      ; VAR_ROM_START <- 1st ROM loc

                ; Set up exception vectors & xcall vectors

_BOOT_EXCV:
        DC      ADDR:(EXC_VECTORS<<SYS_LEN_BITS)|OS_EVECTORS_LENGTH
        MOVE    R0,A1
        DC      ADDR:(OS_EVECTORS_BASE<<SYS_LEN_BITS)|OS_EVECTORS_LENGTH
        MOVE    R0,A2
        DC      OS_EVECTORS_LENGTH
_BOOT_EXCV_LOOP:
        BZ      R0,^_BOOT_XCALLV
        SUB     R0,1,R0
        MOVE    [R0,A1],R1
        MOVE    R1,[R0,A2]
        BR      ^_BOOT_EXCV_LOOP

_BOOT_XCALLV:
        DC      ADDR:(XCALL_VECTORS<<SYS_LEN_BITS)|OS_XVECTORS_LENGTH
        MOVE    R0,A1
        DC      ADDR:(OS_XVECTORS_BASE<<SYS_LEN_BITS)|OS_XVECTORS_LENGTH
        MOVE    R0,A2
        DC      OS_XVECTORS_LENGTH
_BOOT_XCALLV_LOOP:
        BZ      R0,^_BOOTSTACKS
        SUB     R0,1,R0
        MOVE    [R0,A1],R1
        MOVE    R1,[R0,A2]
        BR      ^_BOOT_XCALLV_LOOP

                ; Set up stacks

_BOOTSTACKS:
        DC      0                               ; R0 <- 0
        WRITER  R0,SP
        WRITER  R0,SP'

                ; Invalidate Queue registers

_BOOT1:

        DC      ADDR:SYS_INVADR|(OS_QUEUE0_BASE<<SYS_LEN_BITS)|OS_QUEUE0_MASK
        WRITER  R0,QBM
        DC      ADDR:(OS_QUEUE0_BASE<<SYS_LEN_BITS)
        WRITER  R0,QHL
        DC      ADDR:SYS_INVADR|(OS_QUEUE1_BASE<<SYS_LEN_BITS)|OS_QUEUE1_MASK
        WRITER  R0,QBM'
        DC      ADDR:(OS_QUEUE1_BASE<<SYS_LEN_BITS)
        WRITER  R0,QHL'

                ; Set up XLATE cache

_BOOT3:
        DC      ADDR:(OS_CACHE_BASE<<SYS_LEN_BITS)|OS_CACHE_MASK
        WRITER  R0,TBM

                ; Initialize OS variables

        DC      OS_LOCKED_BASE+OS_LOCKED_LENGTH ; R0 <- Initial heap base
        MOVE    R0,R2                           ; Copy to R2
        DC      VAR_HEAP_BASE                   ; R0 <- Offset to HEAP_BASE var
        MOVE    R2,[R0,A0]                      ; Store in VAR_HEAP_BASE
        DC      VAR_FREETOP                     ; R0 <- Offset to FREETOP var.
        MOVE    R2,[R0,A0]                      ; Store in VAR_FREETOP
        DC      VAR_ROM_START                   ; R0 <- Offset to ROM_START var
        MOVE    [R0,A0],R1                      ; R1 <- First ROM location
        DC      OS_INITIAL_BRAT_LENGTH          ; R0 <- Initial size of BRAT
        MOVE    R0,R2                           ; Copy length to R2
        SUB     R1,R0,R1                        ; R1 <- Base of BRAT
        DC      VAR_BRAT_BASE                   ; R0 <- Offset to BRAT_BASE var
        MOVE    R1,[R0,A0]                      ; Store in VAR_BRAT_BASE
        DC      VAR_BRAT_LENGTH                 ; R0 <- Offset to BRAT_LEN var
        MOVE    R2,[R0,A0]                      ; Store len in VAR_BRAT_LENGTH
        DC      OS_INITIAL_BRAT_MASK            ; R0 <- Initial BRAT hash mask
        MOVE    R0,R2                           ; Move to R2
        DC      VAR_BRAT_HASH_MASK              ; R0 <- Offset to hash mask
        MOVE    R2,[R0,A0]                      ; Put initial hash mask in var
        DC      VAR_NEXT_ID                     ; R0 <- Offset to NEXT_ID var
        MOVE    R0,R2                           ; Copy to R2 for safe keeping
        MOVE    0,R0                            ; R0 <- 0
        MOVE    R0,[R2,A0]                      ; VAR_NEXT_ID <- 0
        DC      VAR_LAST_ID                     ; R0 <- Offset to LAST_ID var
        MOVE    R0,R2                           ; Copy to R2 for safe keeping
        DC      SYS_ID_ID_MASK                  ; R0 <- ID field mask
                                                ;  (same as last ID)
        MOVE    R0,[R2,A0]                      ; Put last ID in VAR_LAST_ID

        DC      VAR_MCACHE_BASE                 ; R0 <- Offset to mcache var
        MOVE    R0,R1                           ; Swap to R1
        DC      OS_MCACHE_BASE                  ; R0 <- Initial base value
        MOVE    R0,[R1,A0]                      ; Set MCACHE_BASE variable
        DC      VAR_MCACHE_LENGTH               ; R0 <- Offset to mcache length
        MOVE    R0,R1                           ; Swap to R1
        DC      OS_MCACHE_LENGTH                ; R0 <- Initial length value
        MOVE    R0,[R1,A0]                      ; Set MCACHE_LENGTH variable
        DC      VAR_MCACHE_OVERFLOW_LIST        ; R0 <- Addr of oflow list
        MOVE    NIL,R1                          ; R1 <- NIL
        MOVE    R1,[R0,A0]                      ; Set oflow list to NIL

                ; Fill Context free list with a few contexts

BOOT_CFREE_INIT:
        MOVE    3,R3                            ; R2 <- Number of ctxts to make
        MOVE    NIL,R0                          ; R0 <- NIL
        PUSH    R0                              ; Push NIL on the stack
BOOT_CFREE_INIT_LOOP:
        DC      CONT_NORMAL_SIZE                ; R0 <- Size of normal context
        CALL    TRAP_NEW_CONTEXT                ; A1 <- New context address
        MOVE    [OBJECT_ID,A1],R1               ; R1 <- Context ID
        POP     R2                              ; R2 <- Old cfree list
        PUSH    R1                              ; Push new cfree list
        MOVE    R2,[CONT_NEXT_CONTEXT,A1]       ; Next context = Old cfree list
        SUB     R3,1,R3                         ; Decrement ctxts left to make
        BNZ     R3,^BOOT_CFREE_INIT_LOOP        ; Loop
        DC      VAR_CFREE_LIST                  ; R0 <- Offset to cfree list
        POP     R1                              ; R1 <- Cfree list
        MOVE    R1,[R0,A0]                      ; Set up Cfree list variable

                ; Enable message reception by masking off disable bits

BOOT_ENABLE_QUEUES:
        DC      ~SYS_INVADR                     ; R0 <- All bits BUT the
                                                ;  invalid address bit
        READR   QBM,R1
        AND     R1,R0,R1                        ; Mask off disable bit
        WRITER  R1,QBM
        READR   QBM',R1
        AND     R1,R0,R1                        ; Mask off disable bit
        WRITER  R1,QBM'
        MOVE    FALSE,R0
        WRITER  R0,I
        BR      ^BKGD_EXC
BOOT_END:


;*********************************************************************************
;
;               B A C K G R O U N D   L O O P S
;
;*********************************************************************************

DIE_TRP:
        BR      ^DIE_TRP
EMPTY_FAULT:
        BR      ^EMPTY_FAULT
EMPTY_TRAP:
        BR      ^EMPTY_TRAP
EMPTY_XCALL:
        BR      ^EMPTY_XCALL
PUSH_EXC:
        BR      ^PUSH_EXC
POP_EXC:
        BR      ^POP_EXC
BKGD_EXC:
```

[illegible] [illegible]_EXC

```
;*************************************************************************
;
;            P R I M I T I V E   M E S S A G E S
;
;*************************************************************************

;-------------------------------------------------------------------------
; WRITE_MSG -- Message routine to write a block of data to consecutive
;       locations.
;
; WRITE (destination-address) (data)*
;

WRITE_MSG:
        MOVE    [1,A3],R0                       ; R0 <- Destination address
        MOVE    R0,A2                           ; Move to A2
        DC      SYS_LEN_MASK                    ; R0 <- Mask to keep len bits
        MOVE    [0,A3],R2                       ; R2 <- message header
        WTAG    R2,TAG_INT,R2                   ; Cast header into an INT
        AND     R0,R2,R2                        ; R2 <- message length
        MOVE    2,R0                            ; R0 <- Src offset into queue
        MOVE    0,R1                            ; R1 <- Dest offset into A2
_WRITE_MSG1:
        GE      R0,R2,R3                        ; Are we at the end of message?
        BT      R3,^_WRITE_MSG_EXIT             ; If so, exit
        MOVE    [R0,A3],R3                      ; Get a "hunk o' data"
        MOVE    R2,[R1,A2]                      ; Toss it into the destination
        ADD     R0,1,R0
        ADD     R1,1,R1
        BR      ^_WRITE_MSG1
_WRITE_MSG_EXIT:
        SUSPEND
WRITE_MSG_END:
```

```
;------------------------------------------------------------------------
; READ_MSG -- Message routine to read a block of data to consecutive
;       locations.
;
; READ (source-address) (reply-node) (reply-header)
;

READ_MSG:
        MOVE    [1,A3],R1                       ; R1 <- address/len of source
        MOVE    R1,A2                           ; Copy to A2
        SEND    [2,A3]                          ; Send reply node number
        DC      SYS_LEN_MASK                    ; R0 <- Mask to keep length
        AND     R1,R0,R1                        ; R1 <- length
        BNZ     R1,^_READ_MSG0                  ; If length != 0, continue
        SENDE   [3,A3]                          ; If no length, just send hdr
        SUSPEND
_READ_MSG0:
        SUB     R1,1,R1                         ; Convert length to offset
        MOVE    0,R2                            ; Initialize index
        SEND    [3,A3]                          ; Send reply header
_READ_MSG1:
        EQUAL   R1,R2,R0                        ; Is index = final index?
        BT      R0,^_READ_MSG2                  ; If so, use SENDE instead
        SEND    [R2,A2]                         ; Send a word of data
        ADD     R2,1,R2                         ; Increment source index
        BR      ^_READ_MSG1                     ; Loop again
_READ_MSG2:
        SENDE   [R2,A2]                         ; Send final word
        SUSPEND
READ_MSG_END:
```

```
;-----------------------------------------------------------------------
; CALL_MSG -- Message routine to run a method
;
; CALL (method-id) (method-specific-args)*
;
;

CALL_MSG:
        MOVE    [1,A3],R2                       ; R2 <- Method-id
        XLATE   R2,R0,XLATE_METHOD              ; R0 <- Method address
        CHECK   R0,TAG_INT,R1                   ; Is this a hint?
        DC      IP:2                            ; IP <- Offset of 2 into method
        PUSH    R0
        POP     IP
CALL_MSG_END:


;-----------------------------------------------------------------------
; SEND_MSG -- Message routine to take an object id, and send the object
;       referenced by the ID the selector "selector-symbol".  If the object
;       is local, the method is run.  If the object is on another node,
;       we forward the message to the node.
;
;  SEND (selector-symbol) (object-id) (args)*
;

SEND_MSG:
        BR      ^SEND_MSG_START                 ; Jump to main code
SEND_MSG_FORWARD_TO_HOME:
        LSH     R1,-SYS_ID_ID_BITS,R1           ; Shift Birthnode number down
        AND     R1,SYS_ID_NODE_MASK,R0          ; Just keep node number field
SEND_MSG_FORWARD_TO_HINT:
        SEND    R0                              ; Send dest. node number
        SUB     R3,1,R3                         ; R3 <- Index to last in queue
        MOVE    0,R0                            ; R0 <- 0
SEND_MSG_FORWARD_LOOP:
        EQUAL   R0,R3,R2                        ; Are we at last item?
        BT      R2,^SEND_MSG_FORWARD_EXIT       ; If so, send with SENDE
        SEND    [R0,A3]                         ; Send item from queue
        ADD     R0,1,R0                         ; Increment R0
        BR      ^SEND_MSG_FORWARD_LOOP
SEND_MSG_FORWARD_EXIT:
        SENDE   [R0,A3]
        SUSPEND
SEND_MSG_START:
        MOVE    [0,A3],R0                       ; R0 <- Message header
        AND     R0,SYS_LEN_MASK,R3              ; R3 <- Length of message
        MOVE    [2,A3],R1                       ; R1 <- Object ID
        XLATE   R1,R0,XLATE_LOCAL               ; R0 <- Bound value of obj ID
        BNIL    R0,^SEND_MSG_FORWARD_TO_HOME    ; If rcvr not here, forward msg
        CHECK   R0,TAG_INT,R2                   ; Is value a hint?
        BT      R2,^SEND_MSG_FORWARD_TO_HINT    ; If so, forward msg to object
        SUB     R3,3,R3                         ; R3 <- Length of args
        MOVE    R0,A2                           ; Copy address to A2
        MOVE    [OBJECT_HDR,A2],R1              ; R1 <- Header of object
        LSH     R1,-SYS_LEN_BITS,R1             ; Shift class down
        AND     R1,SYS_CLASS_MASK,R1            ; R1 <- Class
        DC      SYS_SELECTOR_BITS               ; R0 <- Bits of selector field
        LSH     R1,R0,R1                        ; Shift Class field up
        OR      R1,[1,A3],R1                    ; Merge with selector
        WTAG    R1,TAG_CS,R1                    ; Tag as a class/selector
        XLATE   R1,R2,XLATE_METHOD              ; R2 <- Method ID
        DC      MSG:(CALL_MSG<<SYS_LEN_BITS)    ; R0 <- Msg Header w/o length
        ADD     R3,2,R3                         ; R1 <- Length of CALL message
        OR      R0,R1,R0                        ; Merge with message length
        MOVE    R2,R1                           ; Copy Method-ID to R1
        CALL    TRAP_ID_TO_NODE                 ; R1 <- Node(Method-ID)
        SEND2   R1,R0                           ; Send node, header
        SUB     R3,2,R3                         ; R3 <- Length of args
        BZ      R3,^SEND_MSG_SEND_LAST          ; If no args, just send meth-ID
        SEND    R2                              ; Send Method-ID
        MOVE    3,R0                            ; R0 <- Offset to args
SEND_MSG_LOOP:
        MOVE    [R0,A3],R2                      ; R2 <- Argument from queue
        ADD     R0,T,R0                         ; Increment arg offset
        SUB     R3,1,R3                         ; Decrement lengt/.
        BZ      R3,^SEND_MSG_SEND_LAST          ; If last arg, send & end
        SEND    R2                              ; Send argument
        BR      ^SEND_MSG_LOOP                  ; Loop
SEND_MSG_SEND_LAST:
        SENDE   R2                              ; Send R2 and end
        SUSPEND
SEND_MSG_END:
```

```
;-----------------------------------------------------------------------------
; NEW_METHOD -- Message handler to allocate and fill a method for a given
;       class/selector pair.  This routine calls the InstallMethod handler
;       to make the class/selector/ID bindings, but this routine suspends
;       after calling InstallMethod, without waiting for it to complete.
;
;  NEW_METHOD (class) (selector) (size-of-code) (code)*
;

NEW_METHOD_MSG:
        MOVE    [3,A3],R0                       ; R0 <- Size of code
        ADD     R0,2,R0                         ; Add in 2 header words
        MOVE    CLASS_METHOD,R1                 ; R1 <- "Method" class
        CALL    TRAP_NEW                        ; Allocate an object
        XLATE   R0,A2,XLATE_OBJ                 ; A2 <- Address of object
        MOVE    4,R1                            ; R1 <- Source offset
        MOVE    2,R2                            ; R2 <- Dest offset
        MOVE    [3,A3],R0                       ; R0 <- Size of code
NEW_METHOD_MSG_LOOP:
        BZ      R0,^NEW_METHOD_MSG_INSTALL      ; If no size left then install
        MOVE    [R1,A3],R3                      ; R3 <- Data word
        MOVE    R3,[R2,A2]                      ; Put data word in object
        SUB     R0,1,R0                         ; Decrement size
        ADD     R1,1,R1                         ; Increment source
        ADD     R2,1,R2                         ; Increment destination
        BR      ^NEW_METHOD_MSG_LOOP            ; Loop
NEW_METHOD_MSG_INSTALL:
        MOVE    NNR,R1                          ; R1 <- This node number
        DC      MSG:(CALL_MSG<<SYS_LEN_BITS)|4  ; R0 <- header
        SEND2   R1,R0                           ; Send node,header
        DC      HANDLER_INSTALL_METHOD          ; R0 <- ID of InstallMethod
        SEND    R0                              ; Send InstallMethod ID
        SEND    [1,A3]                          ; Send class
        SEND    [2,A3]                          ; Send selector
        SENDE   [OBJECT_ID,A2]                  ; Send method ID & end
        SUSPEND
NEW_METHOD_MSG_END:
```

```
;-----------------------------------------------------------------------
; NEW_MSG -- Message routine to create a new instance of a certain class and
;       mail back the ID.
;
;  NEW (size-of-object) (class) (reply-id) (reply-selector) (optional-data)*
;

NEW_MSG:
        MOVE    [1,A3],R0                       ; R0 <- length of object
        MOVE    [2,A3],R1                       ; R1 <- class
        CALL    TRAP_NEW                        ; Make a new object
        XLATE   R0,A2,XLATE_OBJ                 ; A2 <- Address of object

                ; *** Copy Optional Data ***

        DC      SYS_LEN_MASK                    ; R0 <- low 10 bit mask
        MOVE    [0,A3],R1                       ; R1 <- Message header
        WTAG    R1,TAG_INT,R1                   ; Cast into an INT
        AND     R0,R1,R0                        ; R0 <- length of message
        SUB     R0,5,R0                         ; Ignore first 5 arguments,
                                                ;  leaving optional data
                                                ;  length in R0
        MOVE    5,R1                            ; R1 <- offset into queue
        MOVE    2,R2                            ; R2 <- offset into object
_NEW_MSG1:
        BZ      R0,^_NEW_MSGEXIT                ; If no data left, exit
        SUB     R0,1,R0                         ; Decrement count
        MOVE    [R1,A3],R3                      ; R3 <- data from msg. stream
        MOVE    R3,[R2,A2]                      ; Store data in object
        ADD     R1,1,R1                         ; Increment offsets
        ADD     R2,1,R2
        BR      ^_NEW_MSG1                      ; Loop
_NEW_MSGEXIT:
        MOVE    [3,A3],R1                       ; R1 <- reply id
        DC      INT:-SYS_ID_ID_BITS             ; R0 <- # of bits of ID
        LSH     R1,R0,R0                        ; Shift node # down & put in R0
        SEND    R0                              ; Send destination node
        DC      MSG:(SEND_MSG<<SYS_LEN_BITS)|4  ; R0 <- SEND message header
        SEND    R0                              ; Mail out the header
        SEND    [3,A3]                          ; Send the target id
        SEND    [4,A3]                          ; Send the selector
        SENDE   [1,A2]                          ; Send new obj ID as final arg
        SUSPEND
NEW_MSG_END:
```

```
;-----------------------------------------------------------------------------
; METHOD_REQUEST_MSG -- Look up a method and mail the ENTIRE method
;       including headers to the requester in a METHOD_REQUEST_REPLY wrapper.
;
; METHOD_REQUEST (method-ID) (reply-node)
;
; Runs under: A0 Absolute-mode, Unchecked
;

METHOD_REQUEST_MSG:
        MOVE    [1,A3],R1                       ; R1 <- Method ID
        MOVE    [2,A3],R2                       ; R2 <- Requester node #
        XLATE   R1,A2,XLATE_METHOD              ; A2 <- Address of method
        DC      SYS_LEN_MASK                    ; R3 <- Mask to keep len field
        AND     R0,A2,R3                        ; R3 <- Length of method
        ADD     R3,2,R3                         ; R3 <- Length of method
                                                ;   + 2 words for msg & ID,
                                                ;  yielding message length
        DC      MSG:(METHOD_REQUEST_REPLY_MSG<<[illegible]
        OR      R0,R3,R0                        ; R0 <- Message header
        SEND2   R2,R0                           ; Send dest node & msg header
        SEND    R1                              ; Send method-ID
        SUB     R3,2,R3                         ; R3 <- Method length
        MOVE    0,R0                            ; Current index = 0
_METHOD_REQUEST_LOOP:
        SUB     R3,1,R3                         ; Decrement length
        BZ      R3,^_METHOD_REQUEST_SEND_LAST   ; If length = 0, send last word
        SEND    [R0,A2]                         ; Mail out method word
        ADD     R0,1,R0                         ; Increment index
        BR      ^_METHOD_REQUEST_LOOP           ; Loop
_METHOD_REQUEST_SEND_LAST:
        SENDE   [R0,A2]                         ; Send final method word
        SUSPEND
METHOD_REQUEST_MSG_END:
```

```
;-----------------------------------------------------------------------------
; METHOD_REQUEST_REPLY_MSG -- Store the method in an object and restart the
;       wait list.
;
; METHOD_REQUEST_REPLY (method-ID) (method-data)*
;
; Runs under: A0 absolute mode, Unchecked
;

METHOD_REQUEST_REPLY_MSG:
        DC      SYS_LEN_MASK                    ; R0 <- Mask to keep length
        AND     R0,[0,A3],R0                    ; R0 <- Length of message
        PUSH    R0                              ; Save R0 on stack
        SUB     R0,2,R0                         ; Ignore message header & ID
        MOVE    CLASS_METHOD,R1                 ; R1 <- Class of a method
        CALL    TRAP_NEW                        ; Make a method object
        XLATE   R0,A2,XLATE_OBJ                 ; A2 <- Address of object
        DC      SYS_COPY_MASK                   ; R0 <- Copy bit
        OR      R0,[OBJECT_HDR,A2],R0           ; R0 <- Hdr marked as a copy
        MOVE    R0,[OBJECT_HDR,A2]              ; Mark object as a copy

        POP     R0                              ; Restore R0 (length of msg)
        SUB     R0,4,R0                         ; R0 <- Len of method w/o hdrs
        MOVE    4,R2                            ; R2 <- Source index
        MOVE    2,R1                            ; R1 <- Destination index
M_R_R_FILL_OBJ:
        BZ      R0,^M_R_R_COPIED                ; If no more length, exit loop
        MOVE    [R2,A3],R3                      ; R3 <- Word from message
        MOVE    R3,[R1,A2]                      ; Put word in method object
        ADD     R1,1,R1                         ; Increment source index
        ADD     R2,1,R2                         ; Increment destination index
        SUB     R0,1,R0                         ; Decrement length left
        BR      ^M_R_R_FILL_OBJ                 ; Loop

M_R_R_COPIED:
        MOVE    [1,A3],R0                       ; R0 <- Original method-ID
        MOVE    A2,R1                           ; R1 <- Method copy address
        ENTER   R0,R1                           ; Enter in XLATE cache
        MOVE    XCALL_BRAT_ENTER_NEW,R3         ; R3 <- BRAT EnterNew Xcall #
        CALL    TRAP_XCALL                      ; Enter in BRAT

        DC      VAR_MCACHE_BASE
        MOVE    [R0,A0],R2                      ; R2 <- Offset to method cache
        DC      VAR_MCACHE_LENGTH
        MOVE    [R0,A0],R3                      ; R3 <- Word size of cache
        MOVE    [1,A3],R1                       ; R1 <- Method ID from message
        ADD     R2,R3,R2                        ; R2 <- Offset past mcache
;
;       Search the Method Cache directory.
;
M_R_R_SEARCH_MC_ID:
        SUB     R2,2,R2                         ; Decrement offset
        SUB     R3,2,R3                         ; Decrement length
        EQ      R1,[R2,A0],R0                   ; Is this the id we want?
        BT      R0,^M_R_R_FOUND_MC_ID           ; If so, branch t
        BNZ     R3,^M_R_R_SEARCH_MC_ID          ; If length != 0, loop
        BR      ^M_R_R_NOT_IN_MCACHE            ; If not in MC, check oflow list
M_R_R_FOUND_MC_ID:
        MOVE    NIL,R0                          ; R0 <- NIL
        MOVE    R0,[R2,A0]                      ; Set ID To NIL
        ADD     R2,1,R2                         ; Point offset to wait list
        MOVE    [R2,A0],R3                      ; R3 <- (car wait-list)
        MOVE    R0,[R2,A0]                      ; Set wait list to NIL
M_R_R_RESTART_CTXT_FROM_MCACHE:
        BNIL    R3,^M_R_R_EXIT                  ; If context ID is nil, exit
        READR   NNR,R2                          ; R2 <- This NNR
        SEND    R2                              ; Send a message to this node
        DC      MSG:(RESTART_CONTEXT_MSG<<SYS_LEN_BITS)|2|SYS_UNC
        SEND    R0                              ; Send message header
        SENDE   R3                              ; Send ID to restart
        XLATE   R3,A2,XLATE_OBJ                 ; Get address of context
        MOVE    [CONT_NEXT_CONTEXT,A2],R3       ; R3 <- next ctxt ID in list
        BR      ^M_R_R_RESTART_CTXT_FROM_MCACHE
M_R_R_EXIT:
        SUSPEND
;
;       If not in MCACHE directory, search overflow list.  Use R2 to hold
;       the previous context ID, and R3 the current context ID.  Use these
;       pointers to delink items from the overflow list.
;
M_R_R_NOT_IN_MCACHE:
        MOVE    NIL,R2                          ; No previous ID
        DC      VAR_MCACHE_OVERFLOW_LIST        ; R0 <- Addr of oflow list
        MOVE    [R0,A0],R3                      ; R3 <- Car of overflow list
M_R_R_LOOP_THRU_OVERFLOW_LIST:
        BNIL    R3,^M_R_R_EXIT                  ; When list NIL, exit
        XLATE   R3,A2,XLATE_OBJ                 ; A2 <- Context Addr
        EQ      R1,[CONT_RESOURCE,A2],R0        ; Waiting for this method?
        BT      R0,^M_R_R_UNLINK_CTXT           ; If so, cut ctxt out of list
        MOVE    R3,R2                              ; Prev ID <- Current ID
        MOVE    [CONT_NEXT_CONTEXT,A2],R3          ; R3 <- next ctxt ID in list
        BR      ^M_R_R_LOOP_THRU_OVERFLOW_LIST
M_R_R_UNLINK_CTXT:
        BNNIL   R2,^M_R_R_UNLINK_MIDDLE_CONTEXT    ; If prev != nil, link to next
M_R_R_UNLINK_FIRST_CONTEXT:
        MOVE    [CONT_NEXT_CONTEXT,A2],R3          ; R3 <- Next context
        DC      VAR_MCACHE_OVERFLOW_LIST           ; R0 <- Addr of oflow list
        MOVE    R3,[R0;A0]                         ; Overflow list <- Next ctxt
        MOVE    R2,[CONT_NEXT_CONTEXT,A2]          ; Next context ptr <- NIL
        MOVE    [OBJECT_ID,A2],R0                  ; R0 <- Ctxt ID
        BR      ^M_R_R_RESTART_CTXT_FROM_LIST      ; Queue up context for execution
M_R_R_LILYPAD:
        BR      ^M_R_R_LOOP_THRU_OVERFLOW_LIST     ; Hop to where we want to be
M_R_R_UNLINK_MIDDLE_CONTEXT:
        MOVE    [CONT_NEXT_CONTEXT,A2],R3          ; R3 <- Next context
        MOVE    NIL,R0                             ; R0 <- NIL
        MOVE    R0,[CONT_NEXT_CONTEXT,A2]          ; Next context <- NIL
        MOVE    [OBJECT_ID,A2],R0                  ; R0 <- ID to clipped-out ctxt
        XLATE   R2,A2,XLATE_OBJ                    ; A2 <- Prev context addr
        MOVE    R3,[CONT_NEXT_CONTEXT,A2]          ; Prev ---> Next (skipping curr)
M_R_R_RESTART_CTXT_FROM_LIST:
        PUSH    R0                                 ; Save context ID
        READR   NNR,R0                             ; R0 <- This NNR
        SEND    R0                                 ; Send a message to this node
        DC      MSG:(RESTART_CONTEXT_MSG<<SYS_LEN_BITS)|2|SYS_UNC
        SEND    R0                                 ; Send message header
        POP     R0                                 ; Restore context ID
        SENDE   R0                                 ; Send ID to restart
        BR      ^M_R_R_LILYPAD                     ; Go to next element in list
METHOD_REQUEST_REPLY_END:
```

```
;-----------------------------------------------------------------
; RESTART_CONTEXT_MSG -- Transfer control to a context
;
; RESTART_CONTEXT (context-id)
;
; Runs under: A0 Absolute mode
;

RESTART_CONTEXT_MSG:
        MOVE    [1,A3],R0                       ; R0 <- Context ID
        CALL    TRAP_XFER_ID                    ; Transfer to context
RESTART_CONTEXT_MSG_END:
```

```
;-----------------------------------------------------------------
; MIGRATE_OBJECT_MSG -- Move an object to a new node
;
; MIGRATE_OBJECT (object-id) (node-number)
;
; Runs under: A0 Absolute mode
;

MIGRATE_OBJECT_MSG:
        MOVE    [1,A3],R0                       ; R0 <- Object ID
        MOVE    [2,A3],R1                       ; R1 <- Dest node number
        MOVE    XCALL_MIGRATE_OBJECT,R3
        CALL    TRAP_XCALL                      ; Migrate the object
        SUSPEND
MIGRATE_OBJECT_MSG_END:

;-----------------------------------------------------------------
; IMMIGRATE_OBJECT_MSG -- Let this object reside on this node
;
; IMMIGRATE_OBJECT (object-id) (previous-residence) (object-data)*
;
; Runs under: A0 Absolute mode, unchecked
;

IMMIGRATE_OBJECT_MSG:
        PUSH    I                               ; Save interrupt status
        MOVE    TRUE,R3                         ; R3 <- True
        MOVE    R3,I                            ; Disable interrupts
        MOVE    [0,A3],R0                       ; R0 <- Message header
        AND     R0,SYS_LEN_MASK,R0              ; R0 <- Message length
        PUSH    R0                              ; Save message length
        SUB     R0,3,R0                         ; R0 <- Object length
        MOVE    [3,A3],R1                       ; R1 <- Object header
        LSH     R1,-SYS_LEN_BITS,R1             ; Shift class down
        AND     R1,SYS_CLASS_MASK,R1            ; R1 <- Class of object
        CALL    TRAP_MALLOC                     ; Mallocate me some memory
        MOVE    [3,A3],R2                       ; R2 <- Object header
        DC      SYS_UNMOVABLE_MASK              ; R0 <- Unmovable bit
        OR      R2,R0,R2                        ; Set unmovable bit in header
        MOVE    R2,[0,A2]                       ; Set header of new object
        MOVE    [1,A3],R0                       ; R0 <- Object ID
        MOVE    A2,R1                           ; R1 <- Address of block
        ENTER   R0,R1                           ; Enter ID/ADDR in XLATE table
        MOVE    XCALL_BRAT_ENTER_NEW,R3         ; R3 <- BRAT EnterNew Xcall #
        CALL    TRAP_XCALL                      ; Enter in BRAT
        MOVE    R0,[1,A2]                       ; Fill 2nd slot with ID
        POP     R0                              ; R0 <- Message length
        SUB     R0,1,R1                         ; R1 <- Offset to last msg word
        SUB     R0,4,R0                         ; R0 <- Offset to end of dest
IMMIGRATE_OBJECT_LOOP:
        EQUAL   R1,4,R2                         ; At first data word?
        BT      R2,^IMMIGRATE_OBJECT_EXIT       ; If so, done
        MOVE    [R1,A3],R2                      ; R2 <- data word
        MOVE    R2,[R0,A2]                      ; Put data word in object
        SUB     R0,1,R0                         ; Decrement R0
        SUB     R1,1,R1                         ; Decrement R1
        BR      ^IMMIGRATE_OBJECT_LOOP          ; Loop
IMMIGRATE_OBJECT_EXIT:
        POP     I                               ; Pop int. disable flag
        DC      MSG:SYS_UNC|(NOW_RESIDING_AT_MSG<<SYS_LEN_BITS)|3
        SEND2   [2,A3],R0                       ; Send previous node #, header
        MOVE    NNR,R0                          ; R0 <- This node number
        SEND2E  [1,A3],R0                       ; Send obj ID and this node #
        SUSPEND
IMMIGRATE_OBJECT_MSG_END:

;-----------------------------------------------------------------
; NOW_RESIDING_AT_MSG -- Notify old residence of new residence & tell birthnode
;
; NOW_RESIDING_AT (object-id) (residence-node)
;
; Runs under: A0 Absolute mode, unchecked.
;

NOW_RESIDING_AT_MSG:
        MOVE    R0,R0                           ; NOP to prevent EARLY Fault
        MOVE    [1,A3],R0                       ; R0 <- Object ID
        MOVE    [2,A3],R1                       ; R1 <- Residence Node #
        ENTER   R0,R1                           ; Cache R0 -> R1
        MOVE    XCALL_BRAT_ENTER,R3             ; R3 <- BRAT_ENTER Xcall #
        CALL    TRAP_XCALL                      ; Bind in BRAT
        MOVE    [1,A3],R1                       ; R1 <- Object ID
        LSH     R1,-SYS_ID_ID_BITS,R1           ; Shift Birthnode number down
        WTAG    R1,TAG_INT,R1                   ; Set tag to INT
        DC      MSG:SYS_UNC|(UPDATE_BIRTHNODE_MSG<<SYS_LEN_BITS)|4
        SEND2   R1,R0                           ; Send header to birthnode
        SEND    [1,A3]                          ; Send object ID
        SEND    [2,A3]                          ; Send new residence node
        MOVE    NNR,R0                          ; R0 <- This node #
        SENDE   R0                              ; Send # as previous residence
        SUSPEND
NOW_RESIDING_AT_MSG_END:

;----------------------------------------------------------------------
; UPDATE_BIRTHNODE_MSG -- Notify the birthnode of the new residence, and
;       mark the object movable
;
; UPDATE_BIRTHNODE (object-id) (residence-node) (previous-node)
;
; Runs under: A0 Absolute mode, unchecked
;

UPDATE_BIRTHNODE_MSG:
        MOVE    NNR,R2                          ; R2 <- This node #
        MOVE    [1,A3],R0                       ; R0 <- Object ID
        MOVE    [2,A3],R1                       ; R1 <- Residence Node #
        MOVE    [3,A3],R3                       ; R3 <- Previous node #
        EQUAL   R3,R2,R2                        ; Was guy previously here?
        BT      R2,^UPDATE_BIRTHNODE_MOVABLE    ; If so, don't rebind again
        ENTER   R0,R1                           ; Cache R0 -> R1
        MOVE    XCALL_BRAT_ENTER,R3             ; R3 <- BRAT_ENTER Xcall #
        CALL    TRAP_XCALL                      ; Bind in BRAT
UPDATE_BIRTHNODE_MOVABLE:
        DC      MSG:SYS_UNC|(OBJECT_MOVABLE_MSG<<SYS_LEN_BITS)|2
        SEND2   R1,R0                           ; Send header to residence
        SENDE   [1,A3]                          ; Send object ID
        SUSPEND
UPDATE_BIRTHNODE_MSG_END:

;----------------------------------------------------------------------
; OBJECT_MOVABLE_MSG -- Mark the object movable
;
; OBJECT_MOVABLE (object-id)
;
; Runs under: A0 Absolute mode, unchecked
;

OBJECT_MOVABLE_MSG:
        MOVE    R0,R0                           ; NOP to prevent EARLY fault
        MOVE    [1,A3],R0                       ; R0 <- Object ID
        XLATE   R0,A2,XLATE_OBJ                 ; A2 <- Object address
        MOVE    [0,A2],R1                       ; R1 <- Object header
        DC      ~SYS_UNMOVABLE_MASK             ; R0 <- All but unmovable bit
        AND     R1,R0,R1                        ; R1 <- Movable object header
        MOVE    R1,[0,A2]                       ; Put header back in object
        SUSPEND
OBJECT_MOVABLE_MSG_END:
```

```
;**************************************************************************
;
;               S Y S T E M   C A L L   T R A P S
;
;**************************************************************************

;-----------------------------------------------------------------------
; XCALL_TRP -- Call an extended system call
;
; Runs under:   A0 absolute mode, unchecked
; Inputs:       R3
; Trashes:      R3
;

XCALL_TRP:
        PUSH    R0                              ; Save R0
        DC      OS_XVECTORS_BASE                ; R0 <- Base of xvectors
        ADD     R0,R3,R3                        ; R3 <- Xvectors + xcall #
        MOVE    [R3,A0],R3                      ; R3 <- Xcall routine IP
        POP     R0                              ; Restore R0
        MOVE    R3,IP                           ; Go to XCALL routine
XCALL_TRP_END:


;-----------------------------------------------------------------------
; SWEEP_TRP -- Sweep all non-marked objects in the heap down
;       towards the base.
;
; Runs under:   A0 shadow
;

SWEEP_TRP:
        BR      ^SWEEP_TRP_START                ; Go to main code
_SWEEP_EXIT:
        DC      VAR_FREETOP                     ; R0 <- @FREETOP
        MOVE    R1,[R0,A0]                      ; FREETOP <- New destination
        POP     I
        POP     R3
        POP     R2
        POP     R1
        POP     R0
        POP     IP
SWEEP_TRP_START:
        PUSH    R0
        PUSH    R1
        PUSH    R2
        PUSH    R3
        DC      VAR_HEAP_BASE                   ; R0 <- Address of HEAP_BASE
        MOVE    [R0,A0],R2                      ; R2 <- Initial source
        MOVE    R2,R1                           ; R1 <- Initial destination
_SWEEP_LOOP:
        PUSH    I
        MOVE    TRUE,R0                         ; R0 <- True
        MOVE    R0,I                            ; Prevent interrupts
        DC      VAR_FREETOP                     ; R0 <- @FREETOP
        MOVE    [R0,A0],R0                      ; R0 <- End of heap
        GE      R2,R0,R0                        ; At or past the end of heap?
        BT      R0,^_SWEEP_EXIT                 ; If so, then exit
_SWEEP_CONTINUE:
        DC      SYS_MARK_MASK                   ; R0 <- Deletion flag mask
        AND     R0,[R2,A0],R0                   ; R0 <- Only deletion bit
        BZ      R0,^_SWEEP_COPY                 ; If not deleted, copy object
        ADD     R2,1,R2                         ; R2 <- Offset to object ID
        MOVE    [R2,A0],R0                      ; R0 <- Object ID
        PURGE   R0                              ; Remove object ID from cache
        MOVE    XCALL_BRAT_PURGE,R3             ; R3 <- BRAT Purge Xcall #
        CALL    TRAP_XCALL                      ; Remove object ID from BRAT
        SUB     R2,1,R2                         ; Make R2 be offset to object
        MOVE    [R2,A0],R0                      ; R0 <- Header of object
        AND     R0,SYS_LEN_MASK,R0              ; R0 <- Length of object
        ADD     R2,R0,R2                        ; Point src to next object
_SWEEP_ITERATE:
        BR      ^_SWEEP_LOOP                    ; Go to next iteration
_SWEEP_COPY:
        MOVE    [R2,A0],R0                      ; R0 <- Header of object
        AND     R0,SYS_LEN_MASK,R0              ; R0 <- Length of object
        ADD     R2,R0,R2                        ; R2 <- End of src
        ADD     R1,R0,R1                        ; R1 <- End of dest
        EQUAL   R1,R2,R3                        ; Does src = dest?
        BT      R3,^_SWEEP_ITERATE              ; If so, go to next object
_SWEEP_COPY_LOOP:
        BNZ     R0,^_SWEEP_COPY_LOOP2           ; If R0 != 0 continue copying
        LSH     R1,SYS_LEN_BITS,R3              ; R3 <- dest_addr << len_bits
        MOVE    [R1,A0],R0                      ; R0 <- Header of object
        AND     R0,SYS_LEN_MASK,R0              ; R0 <- Length of object
        OR      R0,R3,R0                        ; R0 <- base | len
        OR      R0,SYS_REL_MASK,R0              ; Mark R0 as relocatable
        WTAG    R0,TAG_ADDR,R0                  ; Tag as an address
        PUSH    R1                              ; Save R1
```

```
          PUSH     R2                           ; Save R2
          ADD      R1,1,R1                      ; R1 <- dest + 1
          MOVE     [R1,A0],R2                   ; R2 <- ID
          MOVE     R0,R1                        ; R1 <- ADDR:<objdest><objlen>
          MOVE     R2,R0                        ; R0 <- ID
          ENTER    R0,R1                        ; Put ID/ADDR pair in cache
          MOVE     XCALL_BRAT_ENTER,R3          ; R3 <- BRAT Enter Xcall #
          CALL     TRAP_XCALL                   ; Enter in BRAT
          POP      R2                           ; Restore R2
          POP      R1                           ; Restore R1
          MOVE     [R2,A0],R0                   ; R0 <- Header of object
          AND      R0,SYS_LEN_MASK,R0           ; R0 <- Length of object
          ADD      R2,R0,R2                     ; R2 <- Next src
          ADD      R1,R0,R1                     ; R1 <- Next dest
          BR       ^_SWEEP_ITERATE
_SWEEP_COPY_LOOP2:
          SUB      R0,1,R0
          SUB      R1,1,R1
          SUB      R2,1,R2
          MOVE     [R2,A0],R3                   ; Copy a bit o' object
          MOVE     R3,[R1,A0]
          BR       ^_SWEEP_COPY_LOOP
SWEEP_TRP_END:
```

```
;-----------------------------------------------------------------------
; NEW_CONTEXT_TRP -- Create a context for a process
;
; This trap creates a context object when given the size of args
;       and locals in R0.  The context created looks like:
;
;       start + 0:      |___Header___|
;       start + 1:      |_Context-ID_|
;       start + 2:      |PstateOffset| (Offset from Header to pstate)
;       start + 3:      |Next-Context|
;       start + 4:      |__Resource__|
;       start + 5:      |   Space    |  ---
;                       \/\/\/\/\/\/     |
;                                        |  Length of space in R0
;                       |\/\/\/\/\/\/|  ---
;       pstate + 1:     |____ID0_____| (Method ID)
;       pstate + 2:     |____ID1_____|
;       pstate + 3:     |____ID2_____|
;       pstate + 4:     |____ID3_____|
;       pstate + 5:     |_____R0_____|
;       pstate + 6:     |_____R1_____|
;       pstate + 7:     |_____R2_____|
;       pstate + 8:     |_____R3_____|
;       pstate + 9:     |_____IP_____|
;
; The address of the block is returned in A1 & A2.  The accompanying
;       ID registers (ID1 & ID2) are filled with the context ID.  The
;       HEADER & CONTEXT-ID fields are filled in by this routine.  The
;       NEXT-CONTEXT slot is filled with NIL.  It is up to application code
;       to fill in the ID0-3, R0-3, and IP slots since these values may be
;       corrupted while in the system TRAP code.  The PSTATE-OFFSET field is
;       filled in with the offset from the header of the context.  This field
;       can be used to ease the building of a pointer to the pstate portion
;       of context.
;
; If the space needed is <= the normal context size (defined
;       by CONT_NORMAL_SIZE), then a fast context is allocated off of the
;       free list if possible.
;
; Runs under:   A0 absolute mode, unchecked
; Inputs:       R0
; Outputs:      A1,ID1,A2,ID2
; Trashes:      R0
;
;

NEW_CONTEXT_TRP:
        PUSH    R1                              ; Save R1
        PUSH    R2                              ; Save R2
        PUSH    R0                              ; Save R0
        DC      VAR_CFREE_LIST                  ; R0 <- Base of Cfree list
        MOVE    R0,R2                           ; Swap to R2
        POP     R0                              ; Restore R0 with user size
        GT      R0,CONT_NORMAL_SIZE,R1          ; Is size > normal size?
        BT      R1,^NEW_CONTEXT_TRP_ALLOC       ; If so, allocate a new context
        MOVE    [R2,A0],R1                      ; R1 <- 1st ctxt in free list
        BNIL    R1,^NEW_CONTEXT_TRP_ALLOC       ; If no more normal, then alloc
        XLATE   R1,A1,XLATE_OBJ                 ; A1 <- Context Addr
        XLATE   R1,A2,XLATE_OBJ                 ; A2 <- Context Addr
        MOVE    [CONT_NEXT_CONTEXT,A1],R0       ; R0 <- Next Context
        MOVE    R0,[R2,A0]                      ; Point cfree list to next ctxt
        MOVE    NIL,R0                          ; R0 <- NIL
        MOVE    R0,[CONT_NEXT_CONTEXT,A1]       ; Erase next ctxt ptr (for gc)
        POP     R2                              ; Restore R2
        POP     R1                              ; Restore R1
        POP     IP                              ; Return
NEW_CONTEXT_TRP_ALLOC:
        ADD     R0,5,R0                         ; R0 <- Offset to pstate
        PUSH    R0                              ; Save pstate offset
        ADD     R0,CONT_PSTATE_SIZE,R0          ; R0 <- Total context obj size
        MOVE    CLASS_CONTEXT,R1                ; R1 <- "context" class value
        CALL    TRAP_NEW                        ; Make a new object
        XLATE   R0,A2,XLATE_OBJ                 ; A2 <- Address of object
        XLATE   R0,A1,XLATE_OBJ                 ; Copy to A1
        POP     R0                              ; Restore pstate offset
        POP     R2                              ; Restore R2
        POP     R1                              ; Restore R1
        MOVE    R0,[CONT_PSTATE_OFFSET,A2]      ; Fill PSTATE-OFFSET ctxt field
        MOVE    NIL,R0                          ; R0 <- NIL
        MOVE    R0,[CONT_NEXT_CONTEXT,A2]       ; No next context
        POP     IP
NEW_CONTEXT_TRP_END:
```

```
;--------------------------------------------------------------------------
; NEW_TRP -- Trap to generate a new object
;
; Takes the size of the object in R0 and the class in R1 and allocates a block
;       of memory for the object and assigns it a unique ID.  The ID is
;       returned in R0.  The header is tagged as an object header, and the
;       class/length field is filled in.  The ID slot is filled with the
;       newly generated ID for this object.  In addition, the XLATE cache
;       & BRAT are updated.
;
; Runs under:   A0 Absolute mode, Unchecked
; Inputs:       R0,R1
; Outputs:      R0
; Trashes:      R1
;
NEW_TRP:
        PUSH    I                               ; Push int. disable flag
        PUSH    A2                              ; Save A2
        PUSH    R3                              ; Save R3
        MOVE    TRUE,R3                         ; R3 <- True
        MOVE    R3,I                            ; Disable interrupts
        CALL    TRAP_MALLOC                     ; Mallocate me some memory
        LSH     R1,SYS_LEN_BITS,R1              ; Shift class past len bits
        OR      R1,R0,R1                        ; Merge class & length
        WTAG    R1,TAG_OBJHEAD,R1               ; Tag class/length as objheader
        MOVE    R1,[0,A2]                       ; Fill 1st slot with class/len
        CALL    TRAP_GENID                      ; Generate an id into R0
        MOVE    A2,R1                           ; R1 <- Address of block
        ENTER   R0,R1                           ; Enter ID/ADDR in XLATE table
        MOVE    XCALL_BRAT_ENTER_NEW,R3         ; R3 <- BRAT EnterNew Xcall #
        CALL    TRAP_XCALL                      ; Enter in BRAT
        MOVE    R0,[1,A2]                       ; Fill 2nd slot with ID
        POP     R3                              ; Restore R3
        POP     A2                              ; Restore A2
        POP     I                               ; Pop int. disable flag
        POP     IP                              ; Return
NEW_TRP_END:
```

```
;-----------------------------------------------------------------------
; ID_TO_NODE_TRP -- Trap to find the best node number to hope to
;       find an object on.  Enter with the ID of the object in R1
;       and exit with the node number in R1.
;
;
; Runs under:   A0 Absolute mode
; Inputs:       R1
; Outputs:      R1
;

ID_TO_NODE_TRP:
        PUSH    R2
        XLATE   R1,R1,XLATE_ID_TO_NODE
        CHECK   R1,TAG_ADDR,R2                  ; XLATE locally, nil if unbound
        BF      R2,^ID_TO_NODE_EXIT             ; Does tag = ADDR?
ID_TO_NODE_LOCAL:                               ; If not, we are done
        MOVE    NNR,R1
ID_TO_NODE_EXIT:                                ; R1 <- This node number
        POP     R2
        POP     IP                              ; Restore R2
                                                ; Return

;-----------------------------------------------------------------------
; MALLOC_TRP - Primitive memory allocator
;
; Takes length of block to allocate in R0 and allocates a region this
;       size in memory.  The address of the block is returned in A2.
;       If the block couldn't be allocated, A2 is set invalid.  Should
;       be called with interrupts off or a heap_lock flag set.
;
; Runs under: A0 shadow, unchecked
; Input:        R0
; Output:       A2
;

MALLOC_TRP:
        PUSH    R0
        PUSH    R1
        PUSH    R2
        PUSH    R3
        MOVE    R0,R1
        DC      VAR_FREETOP                     ; Copy length to R1
        MOVE    [R0,A0],R2                      ; R0 <- Offset to VAR_FREETOP
        ADD     R2,R1,R3                        ; R2 <- VAR_FREETOP
        DC      VAR_BRAT_BASE                   ; R3 <- address + length
        MOVE    [R0,A0],R0                      ; R0 <- Offset to VAR_BRAT_BASE
        GE      R3,R0,R0                        ; R0 <- Base of BRAT
        BT      R0,^_MALLOC_BAD                 ; Would new block be too big?
        LSH     R2,SYS_LEN_BITS,R0              ; If so, treat it as an error
        OR      R0,R1,R0                        ; Shift freetop base up
        OR      R0,SYS_REL_MASK,R0              ; Merge in the length field
        WTAG    R0,TAG_ADDR,R0                  ; Mark address as relocatable
        MOVE    R0,A2                           ; Cast into an ADDR
        DC      VAR_FREETOP                     ; Copy to A2
        MOVE    R3,[R0,A0]                      ; R0 <- VAR_FREETOP
        POP     R3                              ; Update new freetop
        POP     R2
        POP     R1
        POP     R0
        POP     IP
_MALLOC_BAD:
        CALL    TRAP_DIE
MALLOC_TRP_END:                                 ; Die for now
```

```
;----------------------------------------------------------------------
; FREE_CONTEXT -- Free up the context in ID1
;
; If the size of the context equals the normal fast context size, then
;       we place the context back onto the free list after allocating a
;       new ID for it (in case of late arriving context replies).  Otherwise,
;       the context is marked for deletion.
;
; Runs under:   A0 Absolute Mode
; Input:        ID1
; Trashes:
;

FREE_CONTEXT_TRP:
        PUSH    R0
        PUSH    R1
        MOVE    ID1,R0
        CALL    TRAP_FREE_SPECIFIED_CONTEXT
        POP     R1
        POP     R0
        POP     IP
FREE_CONTEXT_TRP_END:

;----------------------------------------------------------------------
; FREE_SPECIFIED_CONTEXT -- Free up the context specified in R0
;
; If the size of the context equals the normal fast context size, then
;       we place the context back onto the free list after allocating a
;       new ID for it (in case of late arriving context replies).  Otherwise,
;       the context is marked for deletion.
;
; Runs under:   A0 Absolute Mode
; Input:        R0
; Trashes:      R0,R1
;

FREE_SPECIFIED_CONTEXT_TRP:
        PUSH    A2                              ; Save A2
        XLATE   R0,A2,XLATE_OBJ                 ; A2 <- Addr of context
        MOVE    [OBJECT_HDR,A2],R1              ; R1 <- Header of context
        AND     R1,SYS_LEN_MASK,R1              ; R1 <- Length of context
        SUB     R1,4,R1                         ; Subtract 4 first words
        SUB     R1,CONT_PSTATE_SIZE,R1          ; R1 <- User space size
        EQUAL   R1,CONT_NORMAL_SIZE,R1          ; Is user space = normal size?
        BT      R1,^FREE_CONTEXT_TRP_KEEP_HIM   ; If so, add him to the list
        MOVE    [OBJECT_HDR,A2],R1              ; R1 <- Header of context
        OR      R1,SYS_MARK_MASK,R1             ; Set deletion bit
        MOVE    R1,[OBJECT_HDR,A2]              ; Move hdr back to object
        BR      ^FREE_CONTEXT_TRP_EXIT
FREE_CONTEXT_TRP_KEEP_HIM:
;
;       *** No longer need to generate new ID ***
;
;       PURGE   R0                              ; Remove ID R0 from cache
;       PUSH    I
;       PUSH    R3                              ; Save R3
;       MOVE    TRUE,R3                         ; R3 <- True
;       MOVE    R3,I                            ; Disable interrupts
;       MOVE    XCALL_BRAT_PURGE,R3             ; R3 <- Purge Xcall #
;       CALL    TRAP_XCALL                      ; Remove ID from BRAT
;       CALL    TRAP_GENID                      ; Make a new ID
;       MOVE    R0,[OBJECT_ID,A2]               ; Patch new ID into context
;       MOVE    A2,R1                           ; R1 <- Context ADDR
;       ENTER   R0,R1                           ; Make new cache binding
;       MOVE    A2,R1                           ; R1 <- Context Address
;       MOVE    XCALL_BRAT_ENTER,R3             ; R3 <- Enter Xcall #
;       CALL    TRAP_XCALL                      ; Enter binding in BRAT
;       POP     R3                              ; Restore R3
;       POP     I                               ; Restore Interrupts
;       DC      VAR_CFREE_LIST                  ; R0 <- Offset to CFREE list

        MOVE    [R0,A0],R1                      ; R1 <- CFREE base
        MOVE    R1,[CONT_NEXT_CONTEXT,A2]       ; Put CFREE list as next ctxt
        MOVE    [OBJECT_ID,A2],R1               ; R1 <- Object ID
        MOVE    R1,[R0,A0]                      ; CFREE list <- Context ID
FREE_CONTEXT_TRP_EXIT:
        POP     A2                              ; Restore A2
        POP     IP                              ; Return
FREE_SPECIFIED_CONTEXT_TRP_END:
```

```
;--------------------------------------------------------------------------------
; GENID_TRP -- Generate a unique id.
;
; Returns the ID in R0
;
; Runs under:   A0 Absolute Mode.
; Output:       R0
;

GENID_TRP:
        PUSH    R1
        PUSH    R2
        DC      VAR_LAST_ID                     ; R0 <- Offset to LAST_ID
        MOVE    [R0,A0],R1                      ; R1 <- Last ID
        DC      VAR_NEXT_ID                     ; R0 <- Offset to NEXT_ID
        MOVE    [R0,A0],R2                      ; R2 <- Next ID
        GT      R2,R1,R1                        ; Is next ID > last ID?
        BT      R1,^_GENID_BAD                  ; If so, error
        ADD     R2,1,R1                         ; R1 <- next id + 1
        MOVE    R1,[R0,A0]                      ; Increment next id slot
        READR   NNR,R1                          ; R1 <- Node number
        DC      SYS_ID:ID_BITS                  ; R0 <- # of bits in ID field
        LSH     R1,R0,R0                        ; R0 <- NNR shifted past ID
        OR      R0,R2,R0                        ; R0 <- NODE | ID
        WTAG    R0,TAG_OBJID,R0                 ; Tag as an object ID
        BR      ^_GENID_EXIT
_GENID_BAD:
        CALL    TRAP_DIE
_GENID_EXIT:
        POP     R2
        POP     R1
        POP     IP
GENID_TRP_END:
```

```
;-----------------------------------------------------------------------------
; VERSION_TRP -- Return the version number
;
; Returns the version number in R0.  The version number is an INT tagged value
;       where the high 16 bits hold the major version number and the low 16
;       bits hold the minor version number.
;
; Runs under:   A0 Absolute Mode
; Output:       R0
; Trashes:      Internally:     R0
;               Totally:        R0
;
;

VERSION_TRP:
        DC      ROM_VERSION
        MOVE    [R0,A0],R0
        POP     IP
VERSION_TRP_END:


;------------------------------------------------------------------------
; XFERx_TRP -- Transfer execution to a context
;

; The routines XFER_ID_TRP and XFER_ADDR_TRP both transfer control to a context
;       either referenced by virtual or physical pointers.  To transfer by ID
;       enter with ID in R0.  To transfer by address, enter with address in A1.
;       The context is FREEd afterwards.
;
; Runs under:   A0 Absolute Mode
;
; XFER_ID_TRP
; Input:        R0
; Trashes:      Locally:        R0,A0,A1
;               Totally:        R0,A0,A1
;
; XFER_ADDR_TRP
; Input:        A1
; Trashes:      Locally:        R0,A0
;               Totally:        R0,A0
;
; Never returns.
;


XFER_ID_TRP:
        XLATE   R0,A1,XLATE_OBJ                 ; Get context addr in A1
XFER_ADDR_TRP:
        PUSH    I
        MOVE    TRUE,R0                         ; R0 <- True
        MOVE    R0,I                            ; Disable interrupts
        MOVE    [OBJECT_ID,A1],R0               ; R0 <- Context ID
        MOVE    R0,ID1                          ; Set ID1 to context ID
        MOVE    R0,[7,A0]                       ; Store in current context ID
        MOVE    A1,R0                           ; R0 <- Pointer to context
        LSH     R0,-SYS_LEN_BITS,R0             ; Shift addr field down
        ADD     R0,[CONT_PSTATE_OFFSET,A1],R0   ; Add in offset to pstate
        LSH     R0,SYS_LEN_BITS,R0              ; Shift addr field up
        ADD     R0,[CONT_PSTATE_OFFSET,A1],R0   ; Add in pstate length - 1
        ADD     R0,1,R0                         ; R0 <- ADDR:<ps_addr><ps_len>
        MOVE    R0,A1                           ; A1 <- Pointer to pstate
XFER_ADDR_CLR_STACK:
        MOVE    0,R0                            ; R0 <- 0
        WRITER  R0,SP                           ; Flush stack preparing
                                                ;  for context resume
        MOVE    [PSTATE_IP,A1],R0               ; R0 <- Old IP from context
        PUSH    R0                              ; Push IP on stack

        MOVE    [PSTATE_ID0,A1],R0
        WRITER  R0,ID0
        MOVE    [PSTATE_ID2,A1],R0
        WRITER  R0,ID2
        MOVE    [PSTATE_ID3,A1],R0
        WRITER  R0,ID3

        MOVE    [PSTATE_R0,A1],R0
        MOVE    [PSTATE_R1,A1],R1
        MOVE    [PSTATE_R2,A1],R2
        MOVE    [PSTATE_R3,A1],R3

        PUSH    R0                              ; Save R0
        PUSH    R1                              ; Save R1
        MOVE    [OBJECT_ID,A1],R0               ; R0 <- Context ID
        CALL    TRAP_FREE_CONTEXT               ; Free context
        POP     R1                              ; Restore R1
        POP     R0                              ; Restore R0

        INVAL                                   ; Invalidate address regs
        MOVE    ID0,R0                          ; R0 <- Method-ID from context
        BNIL    R0,^XFER_ADDR_CLR_STACK         ; If ID0 slot nil, don't XLATE
        POP     I
        XLATE   R0,A0,XLATE_METHOD              ; R0 <- Address of method
        POP     IP                              ; Transfer execution to context
XFER_ADDR_TRP_END:
XFER_ID_TRP_END:


;--------------------------------------------------------------------------
; BRAT_PEEK_TRP -- Finds the current slot of the ID in the BRAT
;
; Runs under:   A0 Absolute Mode, Unchecked
; Inputs:       R0,R1,A2
; Output:       R0
;
; The ID to hash to give first offset to start searching from is in
;       R0.  R1 holds the actual ID to search for.  A2 holds a pointer to
;       the base of the BRAT table.  R0 and R1 are sometimes different.
;       A time when they would be different would be if you were
;       searching for the slot to put a new value in.  Here R0 would be the
;       new ID since we would want it to be in a proper place.  R1, would
;       hold NIL however, because we are actually looking for an empty slot.
;       When the conditions are met, the offset from the start of the BRAT
;       is returned in R0.  This will always be even.
; If the ID is not in the brat, NIL is returned in R0.
;

BRAT_PEEK_TRP:
        PUSH    R2
        PUSH    R3

                ; Convert the ID into an initial offset key into the BRAT

        WTAG    R0,TAG_INT,R0                   ; Cast R0 into an INT
        LSH     R0,-8,R2                        ; R2 <- ID >> 8
        XOR     R0,R2,R3                        ; R3 <- ID xor (ID >> 8)
        LSH     R2,-8,R2                        ; R2 <- ID >> 16
        XOR     R3,R2,R3                        ; R3 <- R3 xor (ID >> 16)
        LSH     R2,-8,R2                        ; R2 <- ID >> 24
        XOR     R3,R2,R3                        ; R3 <- R3 xor (ID >> 24)
        LSH     R3,1,R3                         ; R3 <- key * 2 = offset
        DC      VAR_BRAT_HASH_MASK              ; R0 <- Offset to Hash Mask
        MOVE    [R0,A0],R0                      ; R0 <- mask
        AND     R3,R0,R3                        ; Now R3 holds key into BRAT

                ; Find the table length

        DC      SYS_LEN_MASK
        AND     R0,A2,R2                        ; R2 <- BRAT length

                ; Search for the ID starting at offset

_BRAT_PEEK_LOOP:
        BZ      R2,^_BRAT_PEEK_FAIL             ; If no more length, fail!
        EQ      R1,[R3,A2],R0                   ; Have we found the target?
        BT      R0,^_BRAT_PEEK_GOT_HIM
_BRAT_PEEK_NEXT:
        SUB     R2,2,R2                         ; Decrement length left
        SUB     R3,2,R3                         ; Decrement current offset
        LT      R3,0,R0                         ; Is Offset < 0?
        BF      R0,^_BRAT_PEEK_LOOP             ; If not, loop

                ; We must wrap around to top of BRAT

        DC      SYS_LEN_MASK
        AND     R0,A2,R3                        ; R3 <- Length of BRAT
        SUB     R3,2,R3                         ; Point to top ID slot in BRAT
        BR      ^_BRAT_PEEK_LOOP

                ; If ID not in table, we end up here

_BRAT_PEEK_FAIL:
        MOVE    NIL,R3                          ; R3 <- NIL
_BRAT_PEEK_GOT_HIM:
        MOVE    R3,R0                           ; R0 <- Offset of ID in BRAT
        POP     R3
        POP     R2
        POP     IP
BRAT_PEEK_TRP_END:
```

```
;$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
;
;               E X T E N D E D    C A L L    R O U T I N E S
;
;$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$


;------------------------------------------------------------------------------
; BRAT_ENTER_XTRP -- Add an ID/ADDR pair to the BRAT
;
; Runs Under:   A0 Absolute Mode, Unchecked Mode
; Inputs:       R0,R1
;
; Takes and ID/ADDR pair in R0 & R1 and enters the pair into the BRAT.
;

BRAT_ENTER_XTRP:
        PUSH    A2
        PUSH    R3
        PUSH    R2
        PUSH    R1
        PUSH    R0

        MOVE    R0,R2                           ; R2 <- ID
        MOVE    R1,R3                           ; R3 <- ADDR

        DC      VAR_BRAT_BASE                   ; R0 <- Offset to BRAT variable
        MOVE    [R0,A0],R1                      ; R1 <- BRAT_BASE
        DC      SYS_LEN_BITS
        LSH     R1,R0,R1                        ; Shift BRAT_BASE to addr field
        DC      VAR_BRAT_LENGTH
        OR      R1,[R0,A0],R1                   ; R1 <- BRAT base | length
        WTAG    R1,TAG_ADDR,R1                  ; Cast R1 into an ADDR
        MOVE    R1,A2                           ; Move BRAT ptr into A2
        MOVE    R2,R0                           ; R0 <- ID that was passed in
        MOVE    R0,R1                           ; R1 <- ID that was passed in
        CALL    TRAP_BRAT_PEEK                  ; Find offset & return in R0
        BNNIL   R0,^_BRAT_ENTER_OK              ; If offset != nil, we got ID
        MOVE    R1,R0                           ; R0 <- ID (still in R1)
        MOVE    NIL,R1                          ; R1 <- NIL
        CALL    TRAP_BRAT_PEEK                  ; Find offset & return in R0
        BNNIL   R0,^_BRAT_ENTER_OK              ; If offset non nil, still room
        CALL    TRAP_DIE                        ; If no room, die for now.
_BRAT_ENTER_OK:
        MOVE    R2,[R0,A2]                      ; Put ID in 1st slot
        ADD     R0,1,R0
        MOVE    R3,[R0,A2]                      ; Put ADDR in 2nd slot

        POP     R0
        POP     R1
        POP     R2
        POP     R3
        POP     A2
        POP     IP
BRAT_ENTER_XTRP_END:
```

```
;------------------------------------------------------------------------
; BRAT_ENTER_NEW_XTRP -- Add a new ID/ADDR pair to the BRAT
;
; Runs Under:    AS Absolute Mode, Unchecked Mode,
; Inputs:        R0,R1
;
; Takes and ID/ADDR pair in R0 & R1 and enters the pair into the BRAT.  The
;       caller must be sure that the ID is not already in the BRAT, because
;       no search is made for pre-existence.  This routine is intended to
;       be a faster way to enter initial bindings, as in a NEW call.
;

BRAT_ENTER_NEW_XTRP:
        PUSH    A2
        PUSH    R3
        PUSH    R1
        PUSH    R0

        PUSH    R0                              ; Save R0
        MOVE    R1,R3                           ; R3 <- ADDR

        DC      VAR_BRAT_BASE                   ; R0 <- Offset to BRAT variable
        MOVE    [R0,A0],R1                      ; R1 <- BRAT_BASE
        DC      SYS_LEN_BITS
        LSH     R1,R0,R1                        ; Shift BRAT_BASE to addr field
        DC      VAR_BRAT_LENGTH
        OR      R1,[R0,A0],R1                   ; R1 <- BRAT base | length
        WTAG    R1,TAG_ADDR,R1                  ; Cast R1 into an ADDR
        MOVE    R1,A2                           ; Move BRAT ptr into A2
        POP     R0                              ; R0 <- ID that was passed in
        MOVE    NIL,R1                          ; R1 <- NIL (find empty slot)
        CALL    TRAP_BRAT_PROBE                 ; Find offset & return in R0
        BNNIL   R0,^_BRAT_ENTER_NEW_OK          ; If offset non nil, still room
        CALL    TRAP_DIE                        ; If no room, die for now
_BRAT_ENTER_NEW_OK:
        POP     R1                              ; R1 <- ID
        PUSH    R1                              ; Push ID back on stack
        MOVE    R1,[R0,A2]                      ; Put ID in 1st slot
        ADD     R0,1,R0
        MOVE    R3,[R0,A2]                      ; Put ADDR in 2nd slot

        POP     R0
        POP     R1
        POP     R3
        POP     A2
        POP     IP
BRAT_ENTER_NEW_XTRP_END:
```

```
;----------------------------------------------------------------------------
; BRAT_XLATE_XTRP -- Xlate an ID from the BRAT into an ADDR
;
; Runs Under: A0 Shadow, Unchecked Mode
; Inputs: R0
; Output: R0
;
; Takes the ID to lookup in the BRAT in R0.  When the corresponding
;       ADDR value is found, it is returned in R0.
;

BRAT_XLATE_XTRP:
        PUSH    A2
        PUSH    R2
        PUSH    R1

        MOVE    R0,R2                           ; R2 <- ID

        DC      VAR_BRAT_BASE                   ; R0 <- Offset to BRAT variable
        MOVE    [R0,A0],R1                      ; R1 <- BRAT_BASE
        DC      SYS_LEN_BITS
        LSH     R1,R0,R1                        ; Shift BRAT_BASE to addr field
        DC      VAR_BRAT_LENGTH
        OR      R1,[R0,A0],R1                   ; R2 <- BRAT base | length
        WTAG    R1,TAG_ADDR,R1                  ; Cast R2 into an ADDR
        MOVE    R1,A2                           ; Move BRAT ptr into A2

        MOVE    R2,R0
        MOVE    R2,R1
        CALL    TRAP_BRAT_PEEK                  ; Find offset & return in R0

        BNIL    R0,^_BRAT_XLATE_RETURN          ; If R0 nil return the nil

        ADD     R0,1,R0
        MOVE    [R0,A2],R0                      ; Pick out ADDR & return in R0
_BRAT_XLATE_RETURN:
        POP     R1
        POP     R2
        POP     A2
        POP     IP
BRAT_XLATE_XTRP_END:
```

```
;-----------------------------------------------------------------------
; BRAT_PURGE_XTRP -- Purge an ID/ADDR pair from the BRAT
;
; Runs Under: A0 Shadow, Unchecked Mode
; Inputs: R0
;
; Enter with ID to purge in R0.  The routine writes NIL into both
;       the ID & ADDR slot of the binding in the table.
;

BRAT_PURGE_XTRP:
        PUSH    A2
        PUSH    R2
        PUSH    R1
        PUSH    R0

        MOVE    R0,R2                           ; R2 <- ID

        DC      VAR_BRAT_BASE                   ; R0 <- Offset to BRAT variable
        MOVE    [R0,A0],R1                      ; R1 <- BRAT_BASE
        DC      SYS_LEN_BITS
        LSH     R1,R0,R1                        ; Shift BRAT_BASE to addr field
        DC      VAR_BRAT_LENGTH
        OR      R1,[R0,A0],R1                   ; R2 <- BRAT base | length
        WTAG    R1,TAG_ADDR,R1                  ; Cast R2 into an ADDR
        MOVE    R1,A2                           ; Move BRAT ptr into A2

        MOVE    R2,R0
        MOVE    R2,R1
        CALL    TRAP_BRAT_PEEK                  ; Find offset & return in R0
        BNIL    R0,^_BRAT_PURGE_RETURN          ; If ID not in table, return

        MOVE    R0,R1
        DC      SYM:0
        MOVE    R0,[R1,A2]
        ADD     R1,1,R1
        MOVE    R0,[R1,A2]

_BRAT_PURGE_RETURN:
        POP     R0
        POP     R1
        POP     R2
        POP     A2
        POP     IP
BRAT_PURGE_XTRP_END:
```

```
;-----------------------------------------------------------------------
; MIGRATE_OBJECT_XTRP -- Takes an object ID and sends object to a node
;
; The ID of the object to migrate is in R0, and the destination node
;       number is in R1.  If the object is not local, a MIGRATE_OBJECT_MSG
;       message is sent to the residence of the object.
;
; Runs under:   A0 absolute mode, unchecked
; Inputs:       R0, R1
; Trashes:      R2, R3
;

MIGRATE_OBJECT_XTRP:
        PUSH    I                               ; Save old I-Disable flag
        MOVE    TRUE,R2                         ; R2 <- True
        MOVE    R2,I                            ; Disable interrupts
        XLATE   R0,R2,XLATE_ID_TO_NODE          ; R2 <- Address of ID in R0
        PUSH    R0                              ; Save ID
        CHECK   R2,TAG_ADDR,R3                  ; Is object local?
        BT      R3,^MIGRATE_OBJECT_LOCAL        ; If so, migrate it
MIGRATE_OBJECT_FORWARD_MESSAGE:
        SEND    R2                              ; Send residence node #
        DC      MSG:(MIGRATE_OBJECT_MSG<<SYS_LEN_BITS)|3
        SEND    R0                              ; Send message header
        POP     R0                              ; Restore object ID
        SEND2E  R0,R1                           ; Send object id & node #
        POP     I                               ; Restore interrupts
        POP     IP                              ; Return
MIGRATE_OBJECT_LOCAL:
        PURGE   R0                              ; Remove binding from cache
        MOVE    XCALL_BRAT_PURGE,R3             ; R3 <- Purge Xcall #
        CALL    TRAP_XCALL                      ; Purge R0 from BRAT
        AND     R2,SYS_LEN_MASK,R3              ; R3 <- Length of object
        DC      MSG:SYS_UNC|(IMMIGRATE_OBJECT_MSG<<SYS_LEN_BITS)
        ADD     R0,R3,R0                        ; Add length of object
        ADD     R0,3,R0                         ; Add 3 for hdr, ID, this node
        SEND2   R1,R0                           ; Send node #, header
        POP     R0                              ; R0 <- ID
        SEND    R0                              ; Send ID
        MOVE    NNR,R0                          ; R0 <- This nodu #
        SEND    R0                              ; Send this node number
        MOVE    0,R0                            ; Current index = 0
MIGRATE_OBJECT_LOOP:
        MOVE    R2,A2                           ; Copy object address to A2
        SUB     R3,1,R3                         ; Decrement length
        BZ      R3,^MIGRATE_OBJECT_LAST         ; If length = 0, send last word
        SEND    [R0,A2]                         ; Mail out object word
        ADD     R0,1,R0                         ; Increment index
        BR      ^MIGRATE_OBJECT_LOOP            ; Loop
MIGRATE_OBJECT_LAST:
        SENDE   [R0,A2]                         ; Send final object word
        DC      TAG_OBJHEAD:SYS_MARK_MASK       ; R0 <- Deletion mark mask
        OR      R0,[0,A2],R0                    ; Mark header deleted
        MOVE    R0,[0,A2]                       ; Store back into header
        POP     I                               ; Restore interrupts
        POP     IP                              ; Return
MIGRATE_OBJECT_XTRP_END:

;$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
;
;                 E X C E P T I O N   H A N D L E R S
;
;$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

;-----------------------------------------------------------------------
; INVADR_EXC -- Exception handler for access of an Ax register with I bit set
;
; Runs under:   A0 absolute mode,unchecked
;

INVADR_EXC:
        PUSH    R0
        PUSH    R1
        PUSH    R2
        PUSH    R3
        MOVE    TRP,R3                          ; R3 <- Faulting instruction
        DC      SYS_OP0_MASK                    ; R0 <- Mask to keep OP0 field
        AND     R3,R0,R2                        ; R2 <- OP0 field
        DC      -(SYS_OP0_BITS + 2 + 2)         ; R0 <- Bits to shift down
        LSH     R3,R0,R1                        ; R1 <- Opcode
        EQUAL   R1,2,R0                         ; Is opcode 2 (READR)?
        BT      R0,^INVADR_EXC_REG_ORIENTED     ; If so, treat OP0 special
        EQUAL   R1,3,R0                         ; Is opcode 3 (WRITER)?
        BT      R0,^INVADR_EXC_REG_ORIENTED     ; If so, treat OP0 special
INVADR_EXC_NORMAL_OP0:
        MOVE    0,R3                            ; R3 <- 0 (means curr. priority)
        DC      %11                             ; Mask to keep Ax bits
        AND     R2,R0,R2                        ; R2 <- A index
        BR      ^INVADR_EXC_REXLATE             ; Re-translate IDx -> Ax
```

```
INVADR_EXC_REG_ORIENTED:
        LSH     R2,-(SYS_OPO_BITS - 1),R3       ; R3 <- Relative priority
        DC      %11                             ; Mask to keep Ax bits
        AND     R2,R6,R2                        ; R2 <- A index
INVADR_EXC_REXLATE:
        LSH     R3,2,R3
        OR      R3,R2,R3                        ; R3 <- (PAA)
INVADR_EXC_DISPATCH_ON_PAA:
        BR      R3                              ; Branch forward R3 words
INVADR_EXC_ID_LOADERS:
        MOVE    ID0,R0                          ; R0 <- ID0
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
        MOVE    ID1,R0                          ; R0 <- ID1
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
        MOVE    ID2,R0                          ; R0 <- ID2
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
        MOVE    ID3,R0                          ; R0 <- ID3
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
        MOVE    ID0',R0                         ; R0 <- ID0'
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
        MOVE    ID1',R0                         ; R0 <- ID1'
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
        MOVE    ID2',R0                         ; R0 <- ID2'
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
        MOVE    ID3',R0                         ; R0 <- ID3'
        BR      ^INVADR_EXC_XLATE               ; Branch and XLATE
INVADR_EXC_XLATE:
        XLATE   R0,R1,XLATE_LOCAL               ; R1 <- Addr, Int, or NIL

;
;       What is object isn't here!  IF XLATE faults, we don't save stacks!
;


;------------------------------------------------------------------------
; EARLY_EXC -- Exception handler for early queue access
;
; Runs under:   A0 shadow
; Trashes:      TEMP0

EARLY_EXC:
        MOVE    R0,[TEMP0,A0]                   ; Save R0 in TEMP0
        POP     R0                              ; R0 <- Return Address
        WTAG    R0,TAG_INT,R0                   ; Cast into an INT
        LSH     R0,-9,R0                        ; Shift R0 to LSBits
        SUB     R0,1,R0                         ; Back up address/phase
        LSH     R0,9,R0                         ; Shift address field back
        WTAG    R0,TAG_IP,R0                    ; Cast back into an IP
        PUSH    R0                              ; Push return IP on stack
        MOVE    [TEMP0,A0],R0                   ; Restore R0
        POP     IP                              ; Retry instruction
EARLY_EXC_END:


;------------------------------------------------------------------------
; SEND_EXC -- Exception handler for send buffer overflow
;
; Runs under:   A0 shadow
; Trashes:      TEMP0

SEND_EXC:
        MOVE    R0,[TEMP0,A0]                   ; Save R0 in TEMP0
        POP     R0                              ; R0 <- Return Address
        WTAG    R0,TAG_INT,R0                   ; Cast into an INT
        LSH     R0,-9,R0                        ; Shift R0 to LSBits
        SUB     R0,1,R0                         ; Back up address/phase
        LSH     R0,9,R0                         ; Shift address field back
        WTAG    R0,TAG_IP,R0                    ; Cast back into an IP
        PUSH    R0                              ; Push return IP on stack
        MOVE    [TEMP0,A0],R0                   ; Restore R0
        POP     IP                              ; Retry instruction
SEND_EXC_END:


;------------------------------------------------------------------------
; XLATE_EXC -- Exception handler for translation fault
;
; Runs under:   A0 Absolute Mode, Unchecked
; Trashes:      TEMP0-4
;

XLATE_EXC:
        MOVE    R0,[TEMP0,A0]                   ; Save data registers in
        MOVE    R1,[TEMP1,A0]                   ;  TEMP0 - TEMP3 for use
        MOVE    R2,[TEMP2,A0]                   ;  as an array
        MOVE    R3,[TEMP3,A0]

        READR   TRP,R0                          ; R0 <- Current priority TRP
        WTAG    R0,TAG_INT,R0
        MOVE    R0,[TEMP4,A0]                   ; TEMP4 <- Current priority TRP
        LSH     R0,-7,R0                        ; Pick out src. register field
        AND     R0,%11,R0
        ADD     R0,TEMP0,R0                     ; Add TEMP0 as start of array
        MOVE    [R0,A0],R0                      ; Load R0 with source ID
        MOVE    R0,R1                           ; Copy ID to R1

        MOVE    XCALL_BRAT_XLATE,R3
        CALL    TRAP_XCALL                      ; See if ID is in BRAT
        BNIL    R0,^XLATE_EXC_NO_BINDING        ; If not, handle no binding

        ENTER   R1,R0                           ; Enter pair in cache

_XLATE_RETRY:
        POP     R3                              ; R3 <- Return IP
        LSH     R3,-9,R3                        ; Shift IP until phase is LSB
        SUB     R3,1,R3                         ; Back up one phase
        LSH     R3,9,R3                         ; R3 <- Failed inst. IP
        PUSH    R3                              ; Put retry IP on stack

        MOVE    [TEMP0,A0],R0                   ; Restore data registers
        MOVE    [TEMP1,A0],R1
        MOVE    [TEMP2,A0],R2
        MOVE    [TEMP3,A0],R3
        POP     IP                              ; Retry failed instruction

XLATE_EXC_NO_BINDING:
        MOVE    [TEMP4,A0],R0                   ; R0 <- Failed instruction
        LSH     R0,-(SYS_OP0_BITS+SYS_OP1_BITS),R2
        DC      (1 << SYS_OP2_BITS) - 1         ; R0 <- mask to keep op2 field
        AND     R2,R0,R2                        ; R2 <- XLATE mode from op2
        EQUAL   R2,XLATE_OBJ,R0                 ; Were we in XLATE_OBJ mode?
        BT      R0,^XLATE_EXC_OBJ_MODE          ; If so, branch
        EQUAL   R2,XLATE_ID_TO_NODE,R0          ; Were we in XLATE_ID_TO_NODE?
        BT      R0,^XLATE_EXC_ID_TO_NODE_MODE   ; If so, branch
        EQUAL   R2,XLATE_METHOD,R0              ; Were we in XLATE_METHOD mode?
        BT      R0,^XLATE_EXC_METHOD_MODE_JUMP  ; If so, branch

XLATE_EXC_LOCAL:                    ; *** Dest must be a data register! ***
        MOVE    TRP,R1                          ; R1 <- Failed XLATE
        DC      %1111111                        ; R0 <- Mask to keep Dest field
        AND     R1,R0,R2                        ; R2 <- Dest field of XLATE
        ADD     R2,TEMP0,R2                     ; R2 <- Temp0[Dest]
        MOVE    NIL,R0                          ; R0 <- NIL
        MOVE    R0,[R2,A0]                      ; Temp0[Dest] <- NIL
        MOVE    [TEMP0,A0],R0                   ; Restore data registers
        MOVE    [TEMP1,A0],R1
        MOVE    [TEMP2,A0],R2
        MOVE    [TEMP3,A0],R3
        POP     IP                              ; Return

XLATE_EXC_OBJ_MODE:
        CALL    TRAP_DIE                        ; Just die for now

XLATE_EXC_METHOD_MODE_JUMP:
        BR      ^XLATE_EXC_METHOD_MODE          ; Jump extender

XLATE_EXC_ID_TO_NODE_MODE:
        MOVE    TRP,R1                          ; R1 <- Failed XLATE
        LSH     R1,-7,R1                        ; Shift Source bits down
        AND     R1,%11,R1                       ; Just keep source bits
        ADD     R1,TEMP0,R1                     ; R1 <- TEMP0 + Rs
        MOVE    [R1,A0],R1                      ; R1 <- Source ID
        LSH     R1,-SYS_ID_ID_BITS,R1           ; Shift Birthnode number down
        AND     R1,SYS_ID_NODE_MASK,R1          ; Just keep node number field
        MOVE    TRP,R2                          ; R2 <- Failed XLATE
        DC      %1111111                        ; R0 <- Mask to keep Dest field
        AND     R2,R0,R2                        ; R2 <- Dest field of XLATE
        ADD     R2,TEMP0,R2                     ; R2 <- TEMP0 + Dest (Rx only!)
        MOVE    R1,[R2,A0]                      ; TEMP[Dest] = birthnode number
        MOVE    [TEMP0,A0],R0                   ; Restore data registers
        MOVE    [TEMP1,A0],R1
        MOVE    [TEMP2,A0],R2
        MOVE    [TEMP3,A0],R3
        POP     IP                              ; Return

XLATE_EXC_METHOD_MODE:
        POP     R3
        LSH     R3,-9,R3                        ; Shift IP until phase is LSB
        SUB     R3,1,R3                         ; Back up one phase
        LSH     R3,9,R3                         ; R3 <- Failed inst. IP

                ; Now R1 holds source ID, & retry IP is in R3

XLATE_EXC_SAVE_MSG:
        PUSH    R1                              ; Save away R1
        PUSH    ID2                             ; Push ID2 on stack

        MOVE    [0,A3],R2                       ; R2 <- Message header
        DC      SYS_LEN_MASK                    ; R0 <- Mask to keep len bits
        AND     R0,R2,R2                        ; R2 <- Length of msg
        ADD     R2,2,R0                         ; R0 <- Length + 2 words hdr
        MOVE    CLASS_MESSAGE,R1                ; R1 <- Class for copied msg
        CALL    TRAP_NEW                        ; Make an object to hold msg.
        XLATE   R0,A2,XLATE_OBJ                 ; A2 <- Address of object
        PUSH    R0                              ; Push msg object ID on stack

        ADD     R2,2,R1                         ; R1 <- Length + 2 words hdr
XLATE_EXC_COPY_MSG:
        BZ      R2,^XLATE_EXC_MAKE_CONTEXT      ; If no length, done copying
        SUB     R2,1,R2                         ; Decrement source index
        SUB     R1,1,R1                         ; Decrement dest index
        MOVE    [R2,A3],R0                      ; R0 <- word from queue
        MOVE    R0,[R1,A2]                      ; Copy into msg object
        BR      ^XLATE_EXC_COPY_MSG             ; Loop

XLATE_EXC_MAKE_CONTEXT:
        MOVE    0,R0                            ; No local space needed
        CALL    TRAP_NEW_CONTEXT                ; A2 <- Context address
        PUSH    I
        MOVE    TRUE,R0                         ; R0 <- True
        MOVE    R0,I                            ; Disable interrupts
        MOVE    A1,R0                           ; R0 <- Pointer to ctxt
        LSH     R0,-SYS_LEN_BITS,R0             ; Shift addr portion down
        ADD     R0,[CONT_PSTATE_OFFSET,A2],R0   ; Add pstate offset to addr
        LSH     R0,SYS_LEN_BITS,R0              ; Shift addr portion back up
        ADD     R0,[CONT_PSTATE_OFFSET,A2],R0   ; Add in length - 1
        ADD     R0,1,R0                         ; R0 <- ADDR:<ps_addr><ps_len>
        MOVE    R0,A2                           ; A2 <- Pointer to pstate

;       A0 -> ????      ID0 -> ????
;       A1 -> Context   ID1 -> Context
;       A2 -> Pstate    ID2 -> ????
;       A3 -> ????      ID3 -> ????

;
;               Fill IP slot of context
;
        MOVE    R3,[PSTATE_IP,A2]               ; Context IP <- backed up IP
;
;               Fill ID slots in context
;
        POP     R3                              ; Point ID3 to msg object
        MOVE    R3,[PSTATE_ID3,A2]
        POP     R3                              ; ID2 is on stack
        MOVE    R3,[PSTATE_ID2,A2]
        READR   ID1,R3
        MOVE    R3,[PSTATE_ID1,A2]
        READR   ID0,R3
        MOVE    R3,[PSTATE_ID0,A2]
;
;               Fill Rx slots in context
;
        MOVE    [TEMP0,A0],R3
        MOVE    R3,[PSTATE_R0,A2]
        MOVE    [TEMP1,A0],R3
        MOVE    R3,[PSTATE_R1,A2]
        MOVE    [TEMP2,A0],R3
        MOVE    R3,[PSTATE_R2,A2]
        MOVE    [TEMP3,A0],R3
        MOVE    R3,[PSTATE_R3,A2]

        CHECK   R1,TAG_CS,R3                    ; Does Tag = class/selector?
        BF      R3,^XLATE_EXC_REQUEST_METHOD    ; If not, we were xlating an id

XLATE_EXC_LOOKUP_METHOD:
        MOVE    NNR,R3                          ; R3 <- This node number
        DC      MSG:(CALL_MSG<<SYS_LEN_BITS)|3  ; R0 <- header
        SEND2   R3,R0                           ; Send node,header
        DC      HANDLER_LOOKUP_METHOD           ; R0 <- ID of LookupMethod code
        SEND2   R0,R1                           ; Send LookupMethod ID,c/s
        SENDE   [OBJECT_ID,A2]                  ; Send context to reply to
        SUSPEND

XLATE_EXC_REQUEST_METHOD:
        DC      VAR_MCACHE_BASE
        MOVE    [R0,A0],R2                      ; R2 <- Base of method cache
        DC      VAR_MCACHE_LENGTH
        MOVE    [R0,A0],R3                      ; R3 <- Length of method cache
        MOVE    NIL,R0
        MOVE    R0,[TEMP4,A0]                   ; TEMP4 <- NIL
        POP     I
        POP     R1                              ; Get R1 back (clean up later)

;               Now R1 holds the method ID, R2 holds the base of
;               the method cache, and R3 holds the length of the
;               method cache

        ADD     R2,R3,R2                        ; R2 <- Offset past mcache
```

```
XLATE_EXC_SEARCH_MC_ID:
        SUB     R2,2,R2                         ; Decrement offset
        SUB     R3,2,R3                         ; Decrement length
        EQ      R1,[R2,A0],R0                   ; Is this the id we want?
        BT      R0,^XLATE_EXC_FOUND_MC_ID       ; If so, add context to list
        MOVE    [R2,A0],R0
        BNNIL   R0,^XLATE_EXC_MC_LOOP           ; If entry not nil, loop again
        MOVE    [TEMP4,A0],R0
        BNNIL   R0,^XLATE_EXC_MC_LOOP           ; If TEMP4 is non-nil, loop
        MOVE    R2,[TEMP4,A0]                   ; Entry is nil, so fill
                                                ;  TEMP4 with offset to this
                                                ;  empty place.

XLATE_EXC_MC_LOOP:
        BNZ     R3,^XLATE_EXC_SEARCH_MC_ID      ; If length != 0, loop
        MOVE    [TEMP4,A0],R0
        BNNIL   R0,^XLATE_EXC_GOT_ROOM          ; If TEMP4 not nil, we found an
                                                ;  empty space in the table.

XLATE_EXC_ENTER_IN_OVERFLOW_LIST:
        MOVE    R1,[CONT_RESOURCE,A2]           ; Resource = Method ID
        DC      VAR_MCACHE_OVERFLOW_LIST        ; R0 <- Overflow list addr
        MOVE    R0,R2                           ; Copy to R2
        MOVE    [R0,A0],R0                      ; R0 <- Car of overflow list
        MOVE    R0,[CONT_NEXT_CONTEXT,A2]       ; Next context = rest of list
        MOVE    [OBJECT_ID,A2],R0               ; R0 <- Context-ID
        MOVE    R0,[R2,A0]                      ; Oflow list <- Context-ID
        BR      ^XLATE_EXC_MAIL_ORDER_METHOD    ; Mail for method

XLATE_EXC_GOT_ROOM:
        MOVE    [TEMP4,A0],R2                   ; R2 <- Empty slot offset
        MOVE    R1,[R2,A0]                      ; Fill MC ID with method ID
XLATE_EXC_FOUND_MC_ID:
        ADD     R2,1,R2                         ; Point offset to wait list
        MOVE    [R2,A0],R0                      ; R0 <- (car wait-list)
        MOVE    [OBJECT_ID,A2],R3               ; R3 <- Context-ID
        MOVE    R3,[R2,A0]                      ; Point wait-list to context
        MOVE    R0,[CONT_NEXT_CONTEXT,A2]       ; Point child slot to the
                                                ;  rest of wait-list (or nil)

                ; Now we have set up the wait list for the method.
                ; We have to mail off a method request to the hometown
                ; node of the method in question (ID in R1).

XLATE_EXC_MAIL_ORDER_METHOD:
        PUSH    R1                              ; Save ID
        CALL    TRAP_ID_TO_NODE                 ; R1 <- Node number of ID
        MOVE    R1,R3                           ; Move to R3
        POP     R1                              ; Restore ID
        DC      MSG:(METHOD_REQUEST_MSG<<SYS_LEN_BITS)|3|SYS_UNC
        SEND2   R3,R0                           ; Send dest node # & message
        READR   NNR,R3                          ; R3 <- This node number
        SEND2E  R1,R3                           ; Send method-ID & this node #
        SUSPEND                                 ; Wait for method reply
XLATE_EXC_END:


;-----------------------------------------------------------------------
;
;

EXC_VECTORS:
        DC      IP:SYS_ABS|(BKPD_EXC<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; DBLFAULT
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; ILGINST
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; ILGADRMD
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; ACCESS
        DC      IP:SYS_ABS|(EARLY_EXC<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; LIMIT
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; INVADR
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; MSG
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; QUEUE
        DC      IP:SYS_ABS|(SEND_EXC<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(XLATC_EXC<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; RANGE
        DC      IP:SYS_ABS|(PUSH_EXC<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(POP_EXC<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; OVERFLOW
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; TYPE
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; IA
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; IB
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; IC
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; ID
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; IE
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)  ; IF
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_FAULT<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(NEW_CONTEXT_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(FREE_CONTEXT_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(XFER_ID_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(XFER_ADDR_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(ID_TO_NODE_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(NEW_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(MALLOC_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(GENID_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(VERSION_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(BRAT_PEEK_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(SWEEP_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(FREE_SPECIFIED_CONTEXT_TRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_TRAP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_TRAP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(XCALL_TRP<<SYS_LEN_BITS)
        DC      IP:(DIE_TRP<<SYS_LEN_BITS)
EXC_VECTORS_END:

XCALL_VECTORS:
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(BRAT_ENTER_XTRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(BRAT_XLATE_XTRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(BRAT_PURGE_XTRP<<SYS_LEN_BITS)
        DC      IP:SYS_UNC|SYS_ABS|(MIGRATE_OBJECT_XTRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|SYS_ABS|(BRAT_ENTER_NEW_XTRP<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
        DC      IP:SYS_ABS|(EMPTY_XCALL<<SYS_LEN_BITS)
XCALL_VECTORS_END:

;-----------------------------------------------------------------------------
; ROM Constants
;

ROM_VERSION:    DC      INT:(1<<16)|0
ROM_SIZE:       DC      INT:(ROM_END - 1024)
TWIDDLE:        DC      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

ROM_END:
        END
```

# Appendix C

# Operating System Quick Reference

# JOSS Quick Reference

## Primitive Message Handlers

| Name | Arguments | Description |
|---|---|---|
| WRITE | (dest-address) (data)* | Fills the block of memory at <dest-address> with the data contained in the message. The <dest-address> word must be a proper ADDR-tagged value. |
| READ | (src-address) (reply-node) (reply-hdr) | Reads the block of memory starting at <src-address> and mails the data back to the <reply-node> in a message whose header is <reply-hdr>. |
| CALL | (method-id) (args)* | Starts up the method with ID <method-id>. The <args> are used by the task being started. |
| SEND | (selector) (receiver-id) (args)* | Starts up the method that performs the operation indicated by <selector> on the object with ID <receiver-id>. The process started uses the <args>. |
| REPLY | (context-ID) (context-slot) (value) | Places a value in the specified slot <context-slot> of the context with ID <context-id>. If the context was waiting for this slot, it will be restarted. |
| NEW_METHOD | (class) (selector) (code)* | Allocates storage for a new method, copies the <code> into the method object, and installs the <class> and <selector> to method ID bindings in the system table. |
| NEW | (size) (class) (id) (selector) (data)* | Allocates a new object of type <class> on a remote node with length <size>, copies the optional <data> into the object, and when done, sends the <selector> to the object with ID <id>. |
| RESTART_CONTEXT | (context-id) | Queues the context with ID <context-id> for execution. |
| MIGRATE_OBJECT | (object-id) (node-number) | Moves the object with ID <object-id> to node number <node-number> |

## System Calls

| Name | Arguments | Description |
|---|---|---|
| XCALL | Xcall routine number in R3 | Calls one of the routines defined in the extended call vector table. This was implemented since the CALL vector table was running out of room. |
| SWEEP | — | Compacts the heap. |
| NEW_CONTEXT | Size of user space in R0 | This routine creates a new context object with R0 words of user space and returns the context address in A1 and A2. R0 is trashed. |
| NEW | Size of object in R0<br>Class of object in R1 | Creates a new object of size R0 and class R1, and returns the object's ID in R0. R1 gets trashed. |
| ID_TO_NODE | Object ID in R1 | Returns a likely node for the object with ID R1 to be on in R1. |
| MALLOC | Block size in R0 | Allocates R0 words of physical memory and returns the address in A2. |
| FREE_CONTEXT | Context ID to free in ID1 | Frees the context with ID in ID1, possibly placing it on the context free list. |
| FREE_SPECIFIED_CONTEXT | Context ID to free in R0 | Frees the context with ID in R0, possibly placing it on the context free list. This trashes R0 and R1. |
| GENID | — | Generates a new ID, and returns the ID in R0. |
| VERSION | — | Returns the OS version number in R0, where the high 16 bits hold the major value, and the low 16 bits the minor value. |
| XFER_ID | Context ID to restart in R0 | Transfers control to the context whose ID is in R0. This never returns. |
| XFER_ADDR | Context address in A1 | Transfers control to the context whose ID is in A1. This never returns. |
| BRAT_PEEK | ID to hash in R0<br>ID to search for in R1<br>Base of BRAT table in A2 | Hashes the ID in R0 to find a first slot in the BRAT to search. A linear search proceeds from there until the ID in R1 is found. When found, the offset from the start of the BRAT where this entry is located is returned. If not found, NIL is returned. |

## Extended System Calls

| Name | Arguments | Description |
|---|---|---|
| BRAT_ENTER | ID to enter in BRAT in R0. Address in R1 | Enters the ID/ADDR pair R0/R1 into the BRAT. |
| BRAT_XLATE | ID to lookup in BRAT in R0. | Looks R0 up in the BRAT and returns the bound value in R0. |
| BRAT_PURGE | ID to purge from BRAT in R0 | Removes the first binding of R0 from the BRAT. |
| MIGRATE_OBJECT | ID of object to migrate in R0. Node to migrate object to in R1 | Migrates the object whose ID is in R0, to the node whose number is in R1. |

# Bibliography

[Aea80] Arvind and et. al. *A Dataflow Architecture with Tagged Tokens.* Technical Report MIT/LCS/TM-174, Massachusetts Institute of Technology, September 1980.

[Dal] W. J. Dally. Joss: the jellybean operating system. Notes from the JOSS Talk.

[Dal86a] W. J. Dally. Directions in concurrent computing. In *Proceedings of the IEEE International Conference on Computer Design*, pages 102–106, October 1986.

[Dal86b] W. J. Dally. *Message-Passing Intermediate Code.* Concurrent VLSI Architecture Group Memo, Massachusetts Institute of Technology, August 1986.

[Dal86c] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures.* PhD thesis, California Institute of Technology, 1986.

[DC] William J. Dally and Andrew A. Chien. Object-oriented concurrent programming in cst. To be presented in the 3rd Symposium on Hypercube Concurrent Computers and Applications.

[Dea87] W. J. Dally and et. al. Architecture of a message-driven processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, June 1987.

[DK87] W. J. Dally and T. F. Knight. *The J Machine: A Concurrent VLSI Message Passing Computer for Symbolic and Numeric Processing.* Concurrent VLSI Architecture Group Memo, Massachusetts Institute of Technology, 1987.

[Hil85] W. Daniel Hillis. *The Connection Machine. An ACM Distinguished Dissertation 1986*, MIT Press, Cambridge, MA, 1985.

[HT87] W. Horwat and B. K. Totty. *Message-Driven Processor Simulator*. Concurrent VLSI Architecture Group Memo, Massachusetts Institute of Technology, December 1987.

[HT88] W. Horwat and B. K. Totty. *Message-Driven Processor Architecture*. Concurrent VLSI Architecture Group Memo, Massachusetts Institute of Technology, 1988.

[Kun82] H. T. Kung. Why systolic arrays? *COMPUTER*, 37–46, January 1982.

[Lam82] B. W. Lampson. Fast procedure calls. In *ACM Symposium on Architectural Support for Operating Systems and Programming Languages*, 1982.

[Lin80] Bruce Lindsay. *Object Naming and Catalog Management for a Distributed Database Manager*. Technical Report RJ2914, IBM Research Laboratory, San Jose, August 29 1980.

[LS80] Bruce Lindsay and Patricia G. Selinger. *Site Autonomy Issues in R*: A Distributed Database Management System*. Technical Report RJ2927, IBM Research Laboratory, San Jose, September 15 1980.

[OSS80] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: an experiment in distributed operating system structure. In *Communications of the ACM*, February 1980.

[RF87] D. A. Reed and R. M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing*. *Scientific Computation Series*, MIT Press, Cambridge, MA, 1987.

[Sed83] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 1983.

[Sei85] C. L. Seitz. The cosmic cube. In *Communications of the ACM*, 1985.

[SFS85] W. Su, R. Faucette, and C. Seitz. *C Programmer's Guide to the Cosmic Cube*. Technical report 5203:TR:85, California Institute of Technology, September 1985.

[Tot87] B. K. Totty. *An OS Kernal for the Jellybean Machine*. Concurrent VLSI Architecture Group Memo, Massachusetts Institute of Technology, August 1987.

[Tot88] B. K. Totty. *Issues of Storage Reclamation in the Jellybean Machine*. Concurrent VLSI Architecture Group Memo, Massachusetts Institute of Technology, January 5 1988.

[Ung87] D. M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System. An ACM Distinguished Dissertation 1986*, MIT Press, Cambridge, MA, 1987.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System. Advanced Computer Science Series*, McGraw-Hill, New York, 1981.