1991

The Wollongong Personal Computer: aspects of software

Gary J. Stafford
University of Wollongong
The Wollongong Personal Computer
Aspects of Software/Hardware Integration

A thesis submitted in fulfilment of the requirements for the award of the degree of

Doctor of Philosophy
(Computer Science)

from

THE UNIVERSITY OF WOLLONGONG

by

Gary J. Stafford, B.Math, M.Math(Waterloo)

Department of Computer Science
1991
I hereby declare that I am the sole author of this thesis. I also declare that
the material presented within is my own work, except where duly
acknowledged, and that I am not aware of any similar work either prior to
this thesis or currently being pursued.

Gary Stafford
Abstract

This thesis describes the design of a small personal computer, which has the hardware design influenced by the requirements of the operating system, and the operating system design influenced by the considerations of hardware efficiency and cost. The operating system has its requirements met with simple hardware support. The hardware is constrained to provide support which can be conveniently used by the operating system.

Various units of the machine are looked at in detail and novel approaches are suggested which will allow software to make best use of the hardware. The video memory board allows multiple pixels to be simultaneously modified, adjacent either horizontally or vertically. The placement of the memory management and cache units allows retention of information across context switches, yet also allows retention of information after memory has been reorganised. Co-processor integration allows programs to be written with no need for knowledge of the existence of co-processors, yet benefit when co-processors are provided, with no emulation overhead.

The operating system is able to handle the assignment of processes to multiple processors, attempting to provide a best fit, even when the processor types may include proper subsets. An interesting approach to compaction is covered, dealing with trying to keep the amount of reorganization to a minimum. A method of supporting communication both with a Send-Receive, and a Send-Receive-Reply scheme is given, which allows all parties to be in control of, or aware of the distinction on a message by message basis.
Acknowledgement

I must thank the members of the department of Computer Science in general. They have been willing to listen to me, and provide valuable input. In specific, let me thank Juris Reinfelds, the Chairman and Professor of the department when I started. Phillip McKerrow has been here for the duration, and his comments have been invaluable. Greg Doherty was willing to wade through this document and find problems and make suggestions which were much appreciated.

I must also thank the numerous students of the subjects I have taught, who were willing to discuss topics apart from what was required to get a pass. These people helped make an environment which was pleasant enough to keep me interested. I would especially like to thank those who were willing to question what I said. These people stand out as bright spots through the years.

Finally I would link to thank my wife and children. My wife has, miraculously, stayed my wife over the time, never once suggesting that if she has to get along without a husband, she might as well not have one. The children have been willing to put up with my coming and going at strange hours, and have never once forgotten who I am.
# Table of Contents

1 Introduction ....................................................................................................1  
   1 Background .................................................................................................1  
   2 Aims ..............................................................................................................2  
   3 Approach ......................................................................................................4  
   4 Relations to Other Works ............................................................................8  

2 Desirable Features ....................................................................................10  
   1 The Operating System ..............................................................................10  
       1 Guiding Considerations ........................................................................11  
       2 Traffic Accidents Happen ..................................................................12  
       3 Zero, One, Two, Infinity .....................................................................14  
   2 Shared Memory vs. Message Passing ......................................................15  
       1 Type of Message Passing ...................................................................18  
       2 Send Blocking .....................................................................................21  
       3 Direct or Indirect Communication .....................................................30  
       4 Receive Primitives ..............................................................................36  
       5 Responses .............................................................................................41  
       6 Message Type ......................................................................................42  
       7 Massive Data Movement ....................................................................45  
   3 The Hardware ...........................................................................................47  
       1 Memory Management ..........................................................................47  
       2 Multiple Processors .............................................................................53  
       3 Cache Handling ....................................................................................53  
       4 Co-processor Integration .....................................................................54  
       5 User Display ..........................................................................................58  
       6 Time of Day Clock ...............................................................................59  
   Summary ........................................................................................................61  

3 Interactive Aspects ....................................................................................63  
   1 Memory Management .................................................................................63  
   2 Cache Memory ............................................................................................77  
   3 Multiple Processors ...................................................................................82  
   4 Bus .................................................................................................................87  
   5 Devices .........................................................................................................88  
   Summary .........................................................................................................90  

4 The Operating System .............................................................................92  
   1 Process Prototypes ....................................................................................92  
       1 The Owner ...........................................................................................93  
       2 The Owner-Driver ...............................................................................94  
       3 The Distributor .....................................................................................95  
       4 The Administrator ...............................................................................96  
       5 The Tradesman ....................................................................................98  
       6 The Receptionist ...............................................................................101  
       7 The Courier ........................................................................................102  
       8 The Notifier ........................................................................................102  
   2 The Kernel ...............................................................................................103  
       1 Shared Segment and Program Management ....................................104  
       2 Process Management ...........................................................................110  
       3 Communication Management ...........................................................113  

(v)
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Program Management</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>4 File System</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>1 File Naming</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>2 Space Management</td>
<td>165</td>
</tr>
<tr>
<td>5</td>
<td>The Others</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>172</td>
</tr>
<tr>
<td>4</td>
<td>Time Management</td>
<td>122</td>
</tr>
<tr>
<td>5</td>
<td>Name Management</td>
<td>124</td>
</tr>
<tr>
<td>6</td>
<td>Dynamic Data Management</td>
<td>127</td>
</tr>
<tr>
<td>7</td>
<td>Event Management</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>Process Descriptor</td>
<td>131</td>
</tr>
<tr>
<td>9</td>
<td>Process Dispatching</td>
<td>132</td>
</tr>
<tr>
<td>3</td>
<td>Program Management</td>
<td>142</td>
</tr>
<tr>
<td>4</td>
<td>File System</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>1 File Naming</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>2 Space Management</td>
<td>165</td>
</tr>
<tr>
<td>5</td>
<td>The Others</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>172</td>
</tr>
<tr>
<td>5</td>
<td>The Hardware</td>
<td>173</td>
</tr>
<tr>
<td>1</td>
<td>The Bus</td>
<td>173</td>
</tr>
<tr>
<td>2</td>
<td>The Processor Board</td>
<td>186</td>
</tr>
<tr>
<td></td>
<td>1 The MASTER Unit</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>2 The DEVICE Unit</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>3 The ROUTE &amp; COUNT Unit</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>4 The Memory Management Unit</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>5 The CACHE Unit</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>6 The PROCESSOR Unit</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>1 The Instruction Set</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>The LOAD/STORE Instruction</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>The LOAD ADDRESS Instruction</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>The FLYING LEAP Instruction</td>
<td>205</td>
</tr>
<tr>
<td></td>
<td>The CALL/JUMP Instruction</td>
<td>206</td>
</tr>
<tr>
<td></td>
<td>The HOP Instruction</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>The IF Instruction</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>The ALU Instruction</td>
<td>210</td>
</tr>
<tr>
<td></td>
<td>The SWITCH Instruction</td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>2 Internal Instruction Representation</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>3 Internal Organization</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>Communications Ring</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td>Command Launch Unit</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>Instruction Fetch Unit</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>Arithmetic and If Unit and Data Access Unit</td>
<td>245</td>
</tr>
<tr>
<td>7</td>
<td>A CO-PROCESSOR Unit</td>
<td>247</td>
</tr>
<tr>
<td>8</td>
<td>Kernel Processor Board</td>
<td>252</td>
</tr>
<tr>
<td>3</td>
<td>The User Interface Device</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>1 Input Interface Memory</td>
<td>257</td>
</tr>
<tr>
<td></td>
<td>2 Output Interface Memory</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>3 Display Memory</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>4 Output Interface Memory Contents</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>5 Other Parts</td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>6 Process Support</td>
<td>282</td>
</tr>
<tr>
<td>4</td>
<td>The Disk Device</td>
<td>284</td>
</tr>
<tr>
<td>5</td>
<td>The Memory Device</td>
<td>288</td>
</tr>
<tr>
<td>6</td>
<td>The Communication Device</td>
<td>293</td>
</tr>
<tr>
<td>7</td>
<td>Other Devices</td>
<td>295</td>
</tr>
<tr>
<td>8</td>
<td>Packaging</td>
<td>295</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>298</td>
</tr>
</tbody>
</table>
Introduction

Much research has gone into the development of both operating systems, and the hardware they execute on. A large amount of effort has gone into operating system research in an attempt to find a model which is efficiently transportable to multiple hardware platforms. More research and effort has gone into making the implementation of a specific operating system fit a specific hardware platform. During the same time research has gone into identifying the best hardware platforms for specific operating systems. On occasion there have been efforts to develop both hardware and the operating system for it at the same time. That is one of the basic goals of this research.

Section 1.1 Background

This research is based on many years of experience. During the years 1975 to 1984, at the University of Waterloo, a general research project into portability was carried out. The interests and expertise of many persons went into this project. During this time three systems programming languages were designed, and compilers for them implemented. Two operating systems were also developed and evolved over the years. Both operating systems were message passing operating systems, although the flavour of message passing changed with time. The programming languages were linear descendants of the language B, the precursor to C. During this time the operating system, and of course the compilers for the languages, were ported to various machines. These machines included various models from Honeywell, Modcomp, Data General, Control Data, and machines based on the Intel and Motorola processors.

Experience gained during the developments of the languages and compilers, and the porting of the operating system to the various platforms, gave a wide exposure to many implementations of various aspects of both hardware and software. Every machine had admirable aspects, as well as those best not mentioned.

Some hardware, while on cursory view appearing to be admirably suited to a message passing system, turned out to be less than pleasing. Others, which seemed to have little in their favour, were some of the best, because they did not attempt too much.
Since the project involved both operating systems and programming languages, all the support utilities were developed within the project. This led to much experience with structuring applications within a message passing framework.

The duration as well as the size of the project gave ample time and scope for discovering the worst, and best, ways to do many things. In particular the most error prone methods were well tested. This provided strong hints on what not to do in the future.

The future is now.

Section 1.2 Aims

Both the operating system and the hardware it is to execute on are to be defined. These two will be defined in relation to each other. This leads to some difficulty since any change to one will change the environment of the other, and thus have noticeable effects. To clarify the problem somewhat the basic ground rules are needed.

A minimalist approach has to be taken. A complete wish list of a large number of people could be assembled, and used as the basis for design. This would assure that the final goal would never be reached. A further, and more fundamental reason for a minimalist approach, is that nothing is perfect. By keeping the scope as small as possible, modifications are more easily made, since the factors which have to be considered with each change are much fewer.

The operating system will be a message passing system of some sort. Shared memory is probably not going to form an integral part of the model. There will be a small kernel which will form the basis for the operating system. This kernel will be the only "process" which modifies the memory of other processes.

Most services will be supported by a collection of processes dedicated to their task, and independent to a great extent from all others. Disjoint processes imply that the kernel must make some information readily available to many processes in an efficient manner.

The message passing style chosen should be easily extended to work over networks when desired. In particular, since machines on a network may cease to operate at arbitrary times, the message passing scheme must be capable of dealing with the unexpected termination of processes.

Since many processes operate in a stereotyped manner, various styles of process should be identified so that implementation of the system, and future applications, are given useful guidance and prototypes to build from.
Because a large number of processes are envisioned, the state which needs to be saved for each should be as little as possible. This is more a hardware concern than one controlled by the operating system.

The hardware will be a multiple processor machine. Support for heterogeneous processors has to be addressed, both to cover changes in time, and to support a diverse set of uses. The changing of the types and numbers of processors should be manageable by a user with a limited knowledge of computers.

The peripheral boards will assume much of the processing tasks of those peripherals. The bus must support a large number of potential masters due to these active devices. The addition or deletion of a peripheral by a naive user must be possible.

Assigning a process to a processor should be a simple task. Forcing a process from a processor should also be simple. The only "interrupt" a processor need respond to is that which switches processes. Because this is a multiple processor machine there is leeway for some delay in the response to process switching.

Cache memory must deal with the multiple processor environment, but should not require sophisticated (complicated) circuitry. Selective caching has to be possible as occasions arise where caching certain information would be detrimental.

The addition of specific co-processors should be reasonably transparent to the software currently executing. More important, the removal of certain co-processors should not cause problems with software, other than the possible increase in execution time. The introduction of future co-processors, which have yet to be conceived of, should be possible.

The user display should be of reasonable resolution, but must be capable of being updated in arbitrary manner with minimal operations. A major use of the display will be for proportional spaced fonts, so painting rectangles of various size and ratio is a major concern.

The final result will be a machine which will be simple enough to be available to a large number of people. Those with more to spend should be able to obtain a more powerful machine. The hardware should be achievable without straining the state of the art. For example, going to a GaAs processor should be the option of the buyer rather than a requirement to get acceptable performance. The operating system should be capable of implementation by one or two people in under two years of effort.
Section 1.3 Approach

The approach taken here is to consider aspects of operating systems, programming languages and hardware platforms concurrently. This complicates the presentation as, for example, there is no fixed hardware definition when the operating system is described. When viewed from a single perspective the situation seems simple.

Consider memory management. From the operating systems view, paging is desirable as it simplifies internal operations and can lay the problem of inefficient memory utilization at the feet of the applications programmers. From the applications view, virtual memory is desirable as it avoids the need for the programmer to consider how inefficiently the memory is being utilized. From the hardware view, a small number of pages is desirable as this will reduce the component count, and allow minimum overhead in address translation. From the operating systems view, segmentation is desirable as it can support fine grained sharing, and lead to efficient use of memory. From the applications view, segmentation is desirable as it supports a modular approach to development and maintenance. From the hardware view, segmentation is desirable as it can allow more efficient use of memory when the number of divisions is small than paging can. A decision should not be made as to the actual memory management scheme until all views have be considered.

The final result presented here was not achieved by methodical work from an initial position to the given situation. Because of the interactions of the hardware and operating system, the situation can best be visualized as a set of springs inside telescoping tubes which are attached at the ends in some specific manner. The stable extension of one tube is determined by other tubes, many of which are not directly connected to the one being observed. If presented in a simple linear manner it would be necessary to read this thesis multiple times. To avoid this the following summary is given. It should give the flavour of later chapters so that the earlier can be seen in their context.

This thesis contains four major parts. Chapter 2 is, essentially, a wish list. Chapter 3 considers the interactive aspects of the operating system and hardware. Chapter 4 looks more deeply at the operating system. Chapter 5 finally considers details of the hardware.

Chapter 2, avoiding many of the details, considers both the operating system and hardware in general terms. First the desirable features of an operating system are considered. Focus then changes to a reasoned comparison of the benefits and liabilities of shared-memory vs. message-passing models. A close examination of the exact style of
message-passing is made, to determine the exact form of message-passing which is desirable.

After the operating system has been considered for a time, focus changes to the hardware. First, a look is taken at the benefits and costs of both paging and segmentation. Consideration is then given to the interesting features of a multiple processor machine with heterogeneous processors. Given multiple processors, the handling of cache memory is briefly considered. The integration of co-processors into such a multiple processor machine is then discussed, focusing on how the co-processors are to be accessed. The requirements of a high bandwidth user interface device are considered in vague terms. Finally the considerations of a time-of-day clock are covered.

Once Chapter 2 has set the stage, Chapter 3 looks more closely at the interactive aspects of the operating system and the hardware. This is done in five sections.

The first section considers memory management. The buddy scheme is seriously considered, as it has many of the benefits of both paging and segmentation, with few of the costs. Uses for various segments are covered, leading to a tentative assignment of logical segments. The buddy scheme is then looked at in closer detail, and a method of efficiently identifying which segments to move, when compacting is required, is presented.

The second section considers caching. Consideration is given to the various forms of caching possible, and their implications. Justification for a pending-write cache, which works with logical addresses, is given. Identification of which segments can benefit from caching is then done.

The third section looks at multiple processors which may be of diverse types. A scheme which will allow the operating system to identify the appropriate processor or processors from the complete set of processors, for any specific process, is given in a rough form. A means which will allow the operating system to be aware of the existence of the processors available is also covered.

The fourth section looks at the bus. The major consideration at this point is the handling of invalid addresses on the bus. The solution for this is presented in chapter 5.

The fifth section looks at devices on the bus. There are two major points of interaction. The first considered is how the operating system can be aware of the existence of a device. The second considered is how software which is needed for control of the device can be made available.
After Chapter 3 has covered some of the interactive aspects, Chapter 4 focuses on the operating system. This chapter is also split into five sections.

The first section introduces various prototypical processes. This provides the groundwork for an understanding of how multiple processes can interact to form a team which can support some aspect of an operating system.

The second section looks in detail at the kernel of the operating system. The various requests which have to be supported are covered. This is done by looking at seven major groupings of the requests. Each group has data structures and algorithms covered as appropriate. A minimal view is taken for the implementation implied. The major point covered is that of assignment of processes to processors for dispatching. The scheme covered deals with not only processors which are in distinct sets, but also processors which may be in multiple sets.

The third section looks at the processes which manage programs. While outside the kernel of the operating system, this is integral to the system as a whole. An implementation which can use a minimal amount of memory is presented, and the coverage exposes a set of requests which are of use in other system server groups.

The fourth section deals with the file system. Name management is covered in detail and an approach which supports the integration of not only normal file systems but other named services such as network accessible file systems and devices is presented. A scheme which can be used to speed the access to files is covered. The problem of a limited number of accessed files is addressed and solved in an efficient manner. The use of various prototypical processes is shown in this section.

The final section is a brief mention of other processes which combine to form the overall system.

Chapter 5 covers the hardware of the machine. As well, it ties into the previous discussions covered in the previous chapters. There are eight sections to this chapter.

The first section covers the bus of the machine. A method to support multiple bus masters is presented which cleanly supports both constant and occasional users of the bus. Bus arbitration can go on during the useful cycles of the bus. A method of assuring that all addresses on the bus are "valid" is covered. A scheme for handling devices of varying speeds is introduced.
The second section covers the processor board. The simple circuitry needed to support the processor board on the bus is covered. This also specifies how a process can be assigned to a specific processor. The operating systems data structures for describing the segments accessible to a process are reflected directly in how the hardware memory management works. The memory management unit is covered, showing how it is loaded and how it efficiently performs mapping and limit checking. The caching section is presented, with a full description of how it can function successfully as a pending-write cache, and how cache contents can be preserved across process assignments.

The processor is then covered in great detail. The minimal instruction set is shown, and then the internal representation is covered. The ability to fold two external instructions into one internal instruction is demonstrated. The processor chip as a hardware implementation of message-passing concepts is covered. The communications ring at the heart of the processor is defined. This is followed by the examination of some of the more interesting sections of the processor. The section which supports multiple instructions being executed concurrently is fully covered. The instruction fetch section, which also handles process switching is detailed, with attention to the actual method of instruction merging.

The method of integrating co-processors as substitutes for subroutines is introduced and covered in detail. This includes not only how the co-processor detects when it should activate, but also how it returns control to the processor, and how a process can be moved from a processor without a co-processor, to one with a co-processor, and still maintain consistency.

The kernel processor board is introduced and the minor differences from a normal board are detailed. The facilities to support multiple processors are finally tied together in this section.

The third section is dedicated to the user interface device. Most of this section is concerned with the display part. The major emphasis is placed on the organization of the display memory. A scheme which supports multiple pixel modifications in both horizontal and vertical groups is covered. The details necessary to implement such a scheme are presented. Benefits of the scheme are shown by example.

The fourth section covers the mass storage device. It is shown to be a device which implements files rather than disks. Integration of types of storage other than simple disks is
covered. Further examples of prototypical process usage are given when the software necessary to support this device are shown.

The fifth section covers memory boards. A method of supporting multiple write accesses to the memory board, using only one memory cycle, is given. Implications of this method for read accesses are detailed. A simple method for the implementation of virtual memory as a feature of the memory board itself is covered. This is tied back to the method of handling varying speed devices on the bus.

The sixth section covers a communications board as a typical device. It provides the basics to indicate how other devices can be introduced.

The seventh section covers devices in general. It summarizes many points made in passing earlier.

The final section covers packaging. It shows how the full machine can be packaged so that the addition or removal of parts of the machine are trivial. It also addresses styling, while also covering the features necessary to make part of the machine easily transportable while leaving the majority of the machine behind.

Section 1.4 Relations to Other Works

One work which most closely resembles this, is that being carried out on the MONADS-PC [Keedy 86]. The multi-processor configuration appears to be more similar to the SPUR processor [Hill et al. 86] than to such machines as the VMP Multiprocessor [Cheriton et al. 86]. There is no attempt to address the same problems as massively parallel processors such as the BBN Monarch [Rettberg et al. 90]. Similarity can also be seen at a superficial level to the Dragon machine [McCReight 85].

The choice of a message passing paradigm [Gentleman 81], rather than a monitor paradigm [Hoare 74], has been made as the basic method for dealing with interacting processes. It seems simpler to explicitly deal with interactions between processes, than to explicitly deal with keeping processes from interacting excessively.

Cache coherency has been a big topic and has been covered in numerous places. Most deal with how to build physical cache hardware to support such consistency, while some [Briggs 83], deal with how to assure that it does not need to be assisted by hardware. The
approach found here is to assure that the situation where there can be inconsistency does not arise.

Assigning processes to processors in a heterogeneous multi-processor environment has been covered in [Ni 81]. This work deals more with a batch style of scheduling than a process dispatching action. [Ma 82] deals with a homogeneous system, and is attempting to reduce the cost of allocation, however the algorithm discussed there does bear a likeness to that discussed here for assignment of processes to processors.

The FLEX Computer described in [Matelan 85] appears to support the introduction of different processor types into the machine in a manner similar to the machine described here, however it has multiple generic bus interfaces.

The Memory Management Unit described here is different from most currently under discussion. Many of these [Thakkar 86], [Cheriton et al. 88] are intimately concerned with virtual memory models, while attempting to avoid the problems with software managed page tables.
Chapter 2
Desirable Features

Before any actual implementation can be attempted, at least a vague idea of the overall aspects of the final product has to be known. In its simplest form this can be a “wish list”. This chapter will cover some important features of the components of the system while, in general, avoiding any interaction between the components. This is not truly possible, for any single decision can have far reaching implications. The next chapter will deal with these interactions.

Section 2.1 The Operating System

An operating system need not be complex to be useful. In fact the simpler the operating system the more easily it can be understood, and hence the more easily it can be used. By providing a minimal environment which is cleanly defined, and efficiently implemented, an operating system can be used as a basis upon which larger or more convenient environments can be built.

If the system is designed with some firm ideas about its eventual use, these ideas tend to limit the possible uses of the system. A “blinkers” effect takes hold and, the final goal in view, solutions to support the perceived needs tend to be directed only to those perceived needs. Should the perceived needs turn out to be the only needs, this sort of approach can produce solutions which do a reasonably good job of supporting those needs. As in many fields, there is usually a better way to solve a problem than by simply solving the problem. Every specific problem is a special case of a more general problem. Solutions to these more general problems are quite often simpler, more efficient, and more easily understood than the best solutions to the specific problem.

The operating system discussed here attempts to be general. This generality is not attempted by providing a large environment but rather a small one. As a small environment the exact set of tools provided is very important. A large environment provides a complete “socket wrench” set with a socket of every conceivable size. A small environment cannot support such a large set and thus has to be limited. An adjustable wrench is a reasonable solution while a pair of pliers, although in a way usable, is not a good solution. In no way can a minimal system be considered perfect for any given situation. What it does provide is
a framework which does give a reasonable environment where solutions can quickly be provided, and should more specialized tools turn out to be highly desirable, resources are available to support these specific tasks.

2.1.1 Guiding Considerations

Three important points must always be kept in mind when an operating system is being defined. First the system must be designed so that the user of the machine is never forced to restart the machine because of quiet failure of the system. The second point to remember is that no programmer is perfect. The third point to remember is that arbitrary limits should be avoided.

2.1.1.1 System Reliability

Under this general heading can come many problems. There are problems that a system cannot be expected to deal with, such as a program which incorrectly calculates results. There are others for which it must accept responsibility.

By far the most studied problem is that of deadlock. Deadlock prevention, avoidance, detection and recovery are reasonably well understood, and solutions can be provided. The strongest statement possible is that a system must not itself deadlock, nor allow any other processes to deadlock. While an admirable goal this can imply an extremely high cost. This is one instance where a general solution does not seem to be the best approach. The concept of communicating processes [Dijkstra 1965],[Hoare 1978] provides a direction to this problem. If a method can be provided which allows processes to simply and easily avoid deadlock situations, this is an acceptable solution. This method can even be used within the system itself to avoid internal deadlocks. One must be careful to be sure that deadlocks have been avoided and not just made highly unlikely. When considering the hold-and-wait aspect of deadlock, providing more instances than can conceivably be needed will not avoid deadlock, only delay it until more instances than is conceivable are actually needed.

Services should never depend on user written processes for proper operation. Systems exist where a file, once accessed for modification, must be de-accessed before another modification access is allowed. This is an acceptable situation. What is not acceptable is a system where, should the accessing process terminate without de-accessing the file, the file is still considered accessed. A server must not only know that a resource is
accessed, but when it both explicitly, and implicitly, becomes de-accessed. This strong statement can be expensive to support, however it can be weakened slightly without losing any benefits. Only when the resource is required does it matter if it has implicitly been de-accessed. Only when another request for the file which appears to be accessed is made does it matter.

Interaction between unrelated processes should not exist. This is again an admirable goal but again not realistic. The simple example of two processes producing output into files, and actually using all the file storage space available before termination is a case in point. Apart from these pathological problems, most interactions can be avoided. A shared memory system provides many opportunities for such interaction while a message passing system tends to restrict the possibilities to just the pathological ones.

Interaction between related processes should be clean and simple. Given a collection of related processes, understanding the overall behaviour involves understanding the internal behaviour of each process, and the interactions between them. These interactions can be by far the most complex part. A good example comes from classical physics. Given solid bodies in space the interaction is easily understood in terms of gravitational attraction. “Body A is a distance D from body B. Body A is thus accelerated toward body B by ... .” Given two bodies an analytical solution is possible. With three or more the problem becomes too complex to support an analytical solution. The interactions are still simple, but the cumulative effect is beyond comprehension. Changing the velocity of one body slightly can change a stable system to one where one or more bodies achieve escape velocity. This is an inescapable problem in the real universe, but adding one more word to a document file and thus causing the graphics system to switch to inverse display is not acceptable.

2.1.1.2 Traffic Accidents Happen

Given that all programmers were perfect, all programs would be perfect, and hence all processes would operate perfectly. That piece of mythology out of the way, attention has to be turned to the real world. Perfect programmers do not exist. Imperfect programs thus exist, and hence processes do not always operate perfectly.

It was mentioned previously that an imperfect process may not completely announce that it is terminating. For example, if process A is producing output to be consumed by process B, and process A does not indicate to process B that all the output has been produced before it terminates, process B could wait forever for more input which will not
arrive. Some means must be provided so that process B can either find out, or be informed of the fact, that there is never going to be any more input. Of these two choices the first is highly reminiscent of a polling situation while the latter leans toward an event driven situation. Whether polling or event driving is preferred is task specific to a degree, but an event driven solution tends to be more easily understood and dealt with.

Another aspect which is highly desirable is the ability to terminate a process asynchronously. Given a process which operates correctly it can always be made to be responsive to requests for its termination. A process operating incorrectly may never reach the stage where it can attend to this request. A system which provides for asynchronous termination can deal with such problems as infinite loops. Asynchronous termination can also be used as a tool to permit simplifications. An example of such a use was presented in a paper by Beaty and Booth [Beaty 82] where the task of flood filling a region was given to a process which did simply that. It did no interrogation of states to see if it should terminate early. Another process which existed to deal with the human user, upon being informed that the flood fill should be terminated simply destroyed the fill process. No complexity was added to either process to deal with an asynchronous event. This made the code easier to read, understand, and have confidence in. As an added benefit, no time was wasted polling for a situation which “never” happened.

Resources allocated by servers to support user processes should be fixed. Dynamic allocation is a great tool to deal with a problem of unknown size. It does carry with it the burden of handling the situation where the request for more has to be refused. If a server can be assured at the start of processing a request that it has the resources to handle the request, the code which does handle that request is simplified and made more understandable. For example, a buffering process, when having accepted data to buffer and then finding no space to save it in, is in a situation which requires intricate handling. If it could be assured that a request to buffer data only ever came when there was space available, processing would be simplified. This is possible with static allocation.

When looking at a system server there are other aspects which should be considered. First, dynamic allocation carries with it a time penalty. While the time for each allocation is small, the cumulative time may not be. For example, if the graphics process allocates a record for each operation it performs, and it is updating a 1024*1024 bit map pixel by pixel, with a dynamic allocation time overhead of only 10 micro-seconds, the complete updating
operation will take an extra 10 seconds. These small delays, accumulated throughout the system, can amount to a large total delay.

Second, there is the problem of system growth. As each server encounters a new maximum for resource usage, the overall size of the system can grow. This can impact the amount of resources left for non-system processes, and lead to a situation where the machine has to be restarted every so often to allow certain tasks to be completed. This system growth can be eliminated if some form of resource reclamation is possible. Resource reclamation does carry with it its own costs in time, complexity, and restrictions.

If a static solution is available, and can be made transparent to the operation of the user processes that the server process serves, that solution is to be desired.

2.1.1.3 Zero, One, Two, Infinity

Limits have been briefly mentioned above. For situations where no limit need exist, no limit should exist. For example, the number of characters which can be transmitted on a serial line has no intrinsic limit and so none should be provided.

Other situations exist where limits are either required or highly desirable. In programming languages for example, the length of an identifier can be defined to have no limit, however when it comes to actually implementing the language this feature, if truly implemented, can have a serious detrimental effect on the performance of the compiler. This effect is in general not necessary since a reasonable limit on the length of an identifier would permit more efficient operation within the compiler and probably would never be encountered by the user. If the maximum length of an identifier was, say, 1024 characters, no programmers would encounter it, and the compiler would be simplified. This argument holds for operating systems as well as compilers. What is the maximum length of a file name?

Given that a limit should exist, should the limit be small medium or large? A limit which is so large that it is never reached is very desirable. Given the fact that the full path name for a file can consist of an arbitrary number of levels, the name of each level being restricted to 1024 characters is not likely to be encountered.
A limit which usually is not encountered is actually not a reasonable limit. In the domain of programming languages, a maximum of 32 significant characters in an identifier seems reasonable. This is reasonable, provided the definition does not go on to state that any characters after the 32nd will be ignored. Programmers are lead into a false sense of security. Since “all” characters in “all” identifiers they have used have been significant they forget the real rule. When they do have two identifiers which differ only in the “insignificant” characters it never occurs to them that these characters are not significant. A reject-when-buffers-full message passing system with a large supply of buffers will “never” reject and so programmers will get into the habit of not checking. When a rejection is given, it will not be noticed, leading to strange behaviour.

Given that a limit cannot be made so large that it will never be encountered, it is actually a good idea to make it so small that it is always encountered. This is the reasoning behind block-till-delivered message passing systems. The buffer size is zero, which means that the buffers are always full, and the sending process always blocks. This tends to be a form of “know where you stand” operation. People operate more effectively when they know what the rules are, no matter what the rules.

2.1.2 Shared Memory vs. Message Passing

Given that there exists more than one process, and that some of these processes are to interact to perform at least one task, there needs to be some means of communication between processes. There are basically two possibilities. There can exist shared memory with which they communicate, or they can send messages between each other. If shared memory is chosen as the communication media, some form of synchronization primitives have to be chosen and implemented. If message passing is chosen as the communication media, some form of message passing primitives have to be chosen and implemented. On the surface there may seem to be little to guide the choice between the two possibilities [Lauer 78] but a choice has to be made. Both hardware and software aspects should be considered before a decision is made.

From a hardware point of view, for a single processor machine there is little to choose between the two alternatives. A shared memory scheme implies some form of segmentation, whether implemented as actual segmentation or as a set of shared pages. A message passing scheme implies some efficient means of message interchange.
A multiple processor machine has extra complications. A shared memory scheme requires a caching mechanism which possibly can deal efficiently with multiple cached copies of changing data. A message passing scheme, by its very nature as a method of communicating between disjoint processes, can avoid multiple copies of cached data. It does, however, still require an efficient means of message interchange. Ignoring other hardware considerations for the moment, software considerations are also important.

Looking at the choice from a software point of view the two schemes have noticeable differences. When memory is shared the correct internal operation of a process requires that not only its program be correct, but that all programs used by other processes which share memory with it be correct. The existence of one faulty program invalidates the process using it. As well it invalidates the correctness arguments for all other programs which are used by processes sharing memory with it.

A case in point was seen in the Thoth operating system [Cheriton et al. 79] running on the TI/990-10 machine. Thoth was a message passing system, but there existed "teams" of processes which shared memory. One such team was the operating system itself. Everything would work well for long periods of time. Then the line printer would be given large amounts of non-printable characters, causing bizarre behaviour such as multiple page ejects. If the printer was not accidentally told to take itself off line, eventually everything would go back to normal, and stay that way for another long period of time. Investigation found that, indeed, the printer driver had been given a very large character count as the size of the buffer to print. The size was vastly larger than the actual print buffer. Code was inserted to check the size it was given, and display a console message if the number was unreasonably large. The programmers were confident that they could now track down which process had generated the incorrect size. A few days later the same bizarre printer operation happened. Looking at the console, there was no message. Peeking into the data structures used by the printer driver, an unreasonably large size was found. Checking the actual code used by the printer driver showed that the checking code was in place and would have generated a message, but it had not. The conclusion was simple. Between the time the code checked the value, and the time it used the value, some process somewhere in the operating system had changed the value. The time between checking and use was only a few micro-seconds. The value was stored between two other values which were correct. These values were stored on the run time stack of the printer driver process. No other process could access this value by name, so some indirect method of access was wrong. A
A methodical way to find the problem would be to check that all pointers used by all processes within the operating system were indeed always correctly manipulated. Fortunately, before this was even attempted the true cause was detected. The areas allocated to the terminal output drivers for use as run time stacks were actually two words too short. If a user requested the complete deletion of an input line on a terminal, and that line included a tab character, then these extra two words were used. All the output driver stacks followed each other in memory. After the output drivers finished their initial procedures, they never used the first few words of their individual stacks, so the extra space used by most drivers had no visible effect. The stack area of the last output driver was immediately before the stack area for the printer driver. If a person using the last terminal deleted a line with the appropriate contents at exactly the right time, the strange behaviour would manifest. No amount of argument or proof of correctness of the printer driver would be of any use. The problem was based on the fact that interaction between processes occurred not only at the specified interfaces, but also in hidden ways due to the inherent ability of one incorrect process to affect another.

A message passing scheme which does not use shared memory is much easier to deal with. If each process exists within its own disjoint address space, the only way one process can affect another is through a message. Producing a correct process thus only requires producing a correct program. If the value 5 is stored in the variable "y", then "y" will contain the value 5 until the process itself changes it. No other process, no matter how ill formed, is able to change "y". Message passing systems can lead to a much higher confidence factor. The places of interaction between processes are limited to the contents of the messages. No hidden interactions are possible. This point alone is a strong argument for message passing over shared memory but there is a further consideration.

It appears that humans deal with many aspects of life by analogy. If a problem can be translated into an analogous problem which is easier to understand, it will be translated. A system of disjoint processes which communicate by message passing is analogous to a collection of people who communicate to perform a given task. Humans know how to deal with collections of interacting people. It is one of the skills necessary to exist in a society. Allowing this basic human skill to be easily applied to understanding software is a great advantage. An organization is organized for this very reason. It is easier to deal with structured communication rather than a milling mob, no matter how polite and considerate the majority of the mob is.
These two considerations, ease of conceptual understanding, and simpler correctness arguments make message passing the obvious choice. This is not to say that memory should never be shared, just that shared address spaces should not be used unless necessary. Three situations exist. In one, there is no need or desire to share memory. In another, the need to share memory is present. In the final, the desire to share memory is the only argument. The first situation needs no further comment.

There are situations where shared memory is, in general necessary. A case in point is access to device control registers. Both the process controlling the device, and the device itself need access to certain locations. Even in systems which attempt to hide devices by having handlers “send” messages to them, the processor at some point has to communicate with the device. This level of shared memory is quite acceptable since a rigid discipline is required only of those programmers which implement sensitive parts of the system. Programmers exist at all levels of ability. The general programmer need not be (and in general does not want to be) concerned with asynchronous interactions.

Some situations, while not strictly needing shared memory, lead to a desire on the programmers part to use a shared memory solution to the problem. Ten processes, each requiring random access to a large volume of data can be implemented without shared memory by providing each process with its own private copy of the data, and building a method of propagating changes across all copies. Such a solution, while avoiding the problems inherent in a shared memory solution requires much more memory, and introduces potentially crippling complications with change propagation. A shared memory solution for such a problem is intuitively more desirable.

The final conclusion which can be reached is that the operating system should be a message passing system, and also provide some limited means of controlled shared memory.

2.1.3 Type of Message Passing

The heart of a message passing operating system is the passing of messages. As such the decisions made at this point have far reaching implications and so have to be carefully considered.
There is a very large range of options to choose from under the heading of message passing. Indeed, stating that a system is a message passing system states essentially nothing about the system. Some major aspects of a message passing system are; the means of addressing, the blocking characteristics of the send operation, the types of reception facilities, the method used to provide replies to questions, the definition of what a message is, and the facilities provided to deal with massive amounts of communicated data. Each of these is important enough to warrant a detailed discussion but first a few general remarks should be made.

With regard to addressing, a message has to be identified with a destination and a source. Receiving a postcard with no identification of who sent it, and from where, is of little use. Conversely, the chance of your postcard getting to the intended destination is slight if you do not even tell the post office what city you are hoping for. There are supposedly two main camps in the dispute over addressing. There is the direct camp and the indirect camp. The direct camp is supposed to support addressing messages directly to processes, while the indirect camp addresses messages to mailboxes. Arguments against direct addressing are persuasive, but misleading. They have convinced many that indirect addressing is by far the better choice. Identifying a process by a source or compile time value is indeed a problem, but this does not need to imply that an indirect mailbox solution is the only alternative. If a process is directly identified by a run time generated value most arguments against direct addressing are no longer valid.

The blocking characteristics of sending a message are very important. People can exist and work effectively, in the messaging system provided by society with its rather lax definition of blocking, by using complex behaviours both inherent and learned. Forcing all programs to maintain this level of sophistication is asking too much. The blocking of the sending process can range from never blocking all the way to always blocking. This can further be complicated by the use of message buffering.

The reception primitives provided are as important as the sending primitives. The exact nature of these primitives can grossly affect the structuring of communicating processes.

Responding to messages which were questions rather than statements is another important aspect. In general a response has to be provided by a server to either confirm that the message was received, or to provide the answer to a question asked. This need not be a
separate issue in itself since picking a particular stance with respect to blocking send operations may well determine how responses are given. The only major concern here is that servers never block dependent on the actions of client processes. Allowing such blocking can lead to undesirable behaviour. If the send operation is a true non-blocking send operation, the simplest solution may be to have the response sent to the original sending process. If a send operation can block in any conceivable situation, some more complex solution will have to be used.

Another very important aspect is the exact definition of what a message is. There is, again, a wide choice available. The message can be a predetermined size, or a variable size, or of some system defined type, or of any user defined type. The exact form chosen impacts not only the efficiency and simplicity of the message passing implementation but also the efficiency and simplicity of the processes which use the message passing primitives. A further impact, and one often ignored, is seen when the message passing primitives are extended to deal with networks of heterogeneous machines.

Related to the definition of what a message can be, is the definition of where a message can be stored. There are two possibilities. If the message can be stored anywhere in the address space of the sending process, and placed anywhere in the address space of the receiving process, the message has to be copied at least once. This implies that the hardware should support rapid memory to memory block moves. The alternative to this generous approach is to restrict placement of messages to an extent. This restriction can allow the use of memory management hardware to remove the message from the address space of one process and place it in that of another. Given this choice, a decision has to be made about whether there is a net flow of address space from sending process to receiving process or whether this transfer is done by exchange. Apart from the readily apparent differences between a copy solution and the alternative, there is also the subtle difference that the sending process in a copying system does not lose the message sent. In the alternative it does, either by actually losing the ability to address the message area, or by having the message area replaced by that of the receiving process. This subtle difference, when actual use is concerned, can have a great impact on the structure of the processes concerned.

A final area of concern is the movement of massive amounts of data. This is most easily seen in the handling of files. If a process has 40,000 characters to place in a file, some means must be used to pass these characters to the process which eventually places
them on the storage media. The sending and receiving of messages can be used to move the
data. If a copying send has been chosen, this can result in a large number of superfluous
copies, since the original sending process and final receiving process may be separated by
many other processes. Another approach is to use separate primitives to move large masses
of data. If these extra primitives are the chosen method, some form of protection from
incorrect specification of areas has to be provided. If a process tries to have 512 bytes
written to a file, and gets 1024 bytes of its run time stack overwritten the results would not
be as desired.

For convenience during this section, one specific cross section of all of these potential
systems will be chosen as a basis for discussion, and alternatives compared to see if a better
specification can be arrived at. The basis is quite close to that provided by the Port
operating system, a descendant of the Thoth system. The Port scheme for message
passing works well. An interesting aspect is that the Port scheme is itself not so much
designed as evolved. Starting as a simple message passing kernel in 1976 in the original
Thoth operating system, the exact form of message passing primitives shifted around the
spectrum of possibilities as knowledge and experience increased. The Port scheme is the
way it is because that is how it "should" be. The corollary to "If it works, don't fix it" is "If
it can be better, make it better" and that is how Port evolved.

The Port scheme uses a user defined type of message, copied from sending process
to receiving process. The sending process blocks until a reply is provided by the receiving
process. Identification for sending and receiving messages is by a process identifier PID,
which is generated at the time the process is created. Receivers may receive specifically
from a named process or from any process sending to it. Massive data movement is
supported by a separate primitive for transferring data from the address space of one
process to another, provided that one of the processes is the process making the request,
and that the other is blocked waiting for a reply.

**2.1.3.1 Send Blocking**

How send operations are blocked is tightly connected to a buffering strategy. The basic
problem is to move a message from the process sending it, to the process receiving it. The
introduction of buffering in this task can change how the sending process is blocked.
There are four possible approaches to blocking the send operation. The first is that the sending process never blocks. The second is that the sending process blocks only when all buffers are full. The third is blocking the sending process until the message is received. The fourth is blocking the sending process until the message has been received, and a response to the message produced. The initial position taken is that the sending process will block until a response is given. The other three positions need to be examined in some detail. If any of them offer advantages that the chosen one lacks, they will have to be seriously considered.

**Block Until Receive**

Since blocking until reception is the alternative closest to the one chosen it is a reasonable one to consider first. There are two conceivable reasons why a process should be blocked until the message is received, but not until the response is given. One has to do with the utility of a response and the other has to do with the length of the service time.

It can be quite common that there really is no response to the message sent. For example, when a process is finished using a file it sends a message to the file system announcing this fact. This gives the file system a chance to adjust its internal environment as appropriate. Here there is no need for any content in the response the file system would give. Another example occurs in a tracking task. If one process monitors the position of a pointing device and sends new locations to a process which maintains the current known location, the response to, “The pointing device moved over there” can again be empty. In these cases, and in general, blocking until a response is given need not be much of a problem, for the response can be generated as soon as the message sent has been understood, and before any processing implied by the message has been done. The sending process will thus be blocked for, at most a few extra milliseconds. This argument is not strong enough to force a re-evaluation of the chosen blocking rule.

The previous argument is based on one hidden assumption. It is assumed that the machine in question is a uni-processor. In such a case, given that only one process can be active at any one time, when the receiving process is running, the sending process is effectively blocked. When a multiple processor machine is considered, the argument can change drastically. Blocking until reception means that both the sending process and receiving process can be active after the message has been transferred. In the case of servers, they are generally considered of higher priority than the processes they serve (else
they become bottle-necks) and so when the **SEND** occurs, the receiving process is generally waiting for the message. If the sending process, which was active, can continue to be active, there is a reasonable chance that it can be re-assigned to the same processor that it was using. This can allow any cached information in that processor to remain and can result in a noticeable difference. If the sending process blocked until a response was given, it would (for however short a period) not be eligible for a processor. This could result in the receiving process being given the processor which was used by the sending process. This is quite likely since the processor in question is not in use, and a processor is needed for the receiving process which has just become eligible for a processor.

Reassigning the processors in an inefficient way is undesirable. The situation would be even worse if the sending process was more important than the receiving process:
1/ the cache in the processor used by the sending process is emptied  
2/ the receiving process is given the processor  
3/ the cache starts to fill  
4/ the response is given  
5/ the cache is emptied again  
6/ and finally the cache starts to fill back to the level it would have retained if the sending process had not blocked until the response was given.

This argument based on a multiple processor machine is quite convincing but before switching to a block until received operation more consideration has to be given. If responses are never needed the above argument would be complete, but responses are needed and how they are to be implemented needs some thought.

With a system which blocks until reception the server which has to provide a response can be given one of two options. One is to send the response to the original sending process and wait for it to receive it. The other is to provide another primitive to support the transfer of responses. Whichever is chosen it must be remembered that the server should not block waiting upon the whims of the original sending process. If the original sending process asked, “How big is this file?” and never tried to receive the response, the file system would remain blocked, and unavailable to all other processes. If a new primitive were provided for the transfer of responses, that would assure that the respondent would not block, this would be overcome but now a third (and potentially fourth to allow the original sending process to accept the response) primitive would have to be designed. This would essentially be a non-blocking **SEND** and corresponding **RECEIVE**. It would seem to be obvious that
implementing a non-blocking SEND to allow a block until received system to function would be a little excessive. In fact a full non-blocking SEND is not needed since the transfer of a response can be treated as special, as it truly is.

Apart from complexity issues there is a simple aspect worth considering. If it is assumed that all communication primitives take roughly the same amount of time, an expression describing the time used for communication can be given for both the block until receive and block until response systems. Assume that of M messages sent, N require responses. Assume that the SEND, RECEIVE and REPLY primitives all take unit time. Let TSR be the time used in a block until receive system, TSRR be the time used in a block until response system and TD the amount by which TSRR exceeds TSR. Then:

\[
\begin{align*}
\text{TSR} &= (M-N)*2 + (N*4) = M*2 + N*2 \\
\text{TSRR} &= M*3 = M*2 + M \\
\text{and} \\
\text{TD} &= \text{TSRR} - \text{TSR} = (M*2+M)-(M*2+N*2) = M - N*2
\end{align*}
\]

Thus, if the percentage of messages requiring a response is greater than or equal to 50% of the total message count, a block until response system would spend less time in the communication. The assumption that all communication primitives take an equal amount of time is not true. In general the primitive which handles the response in a system which blocks until a response can be much faster than the sending and receiving primitives since it is more tightly constrained. This argument goes against using block-until-receive but ignores the multiple processor effects. It would seem that whichever position is chosen there are good arguments against it.

If it were possible for there to be an identifiable difference between messages which are essentially statements, and those which are questions, it is possible to have both definitions of blocking in one system. Using the same formula notation as above, if the appropriate blocking scheme could be used on an individually chosen basis, the results are slightly better. Let TSRr be the time used if blocking till response were used where appropriate, and blocking till reception were used in the other cases.

\[
\text{TSRr} = (M-N)*2 + N*3 = M*2 + N
\]

Being able to choose uses M-N less primitives than a pure blocking till response solution, and N less primitives than a pure blocking till reception solution.
For a process which sends a “statement”, there is no ready way to tell if it blocked until receive or until response, nor is there any need for it to do so. Any response which may have been given will be treated as “social lubricant”. It is the equivalent of the “Thank you” in human communication. In human affairs whether “Here is the paper you need”, is responded to with “Thank you”, or “About time”, only matters to a sensitive person. Programs being notoriously thick skinned, this is not really relevant.

If the message sent is a “question”, it is vitally important that both the sending process, and the receiving process realize that it is a question. As well, the implementation of the primitives must realize this. The sending process and receiving process will in general understand the difference since the content of the message has to be understood by both. The difficult part is getting the implementation of the primitives to make the distinction since the implementation in general need not “understand” the message in order to transfer it. Whether this distinction should be made by requiring the sending process to request the send in two different ways, or for the format of the message to be partially decipherable by the primitive implementation is a detail that should be left until later.

It was mentioned that there are two conceivable reasons why a process should be blocked until the message is received, but not until the response is given. The first has been adequately covered and now the second needs to be addressed. It is possible that the sending process can do useful work before the response is given even though the response is needed.

A good example of this is the processing of a sequential file. While the first block of the file is being processed the second is being read. This double buffering allows the time taken to read data to be overlapped with the time taken to process it. In a uni-processor machine the time saving can only equal the actual I/O time required to obtain the second block. In a multiple processor machine this overlap can also involve the time the file system uses to compute which block of the storage medium actually contains the required second block. This can amount to a reasonably long time even when efficiently implemented. A case in point was observed with Port when it was first being extended to a network environment. When a program was compiled with the files on a different machine from the one the compiler was executing on, the compilation took 20% less real time. This was while using an inefficient implementation on a relatively slow network. The ability to let compilation continue while the file system was computing gained that much time despite the losses inherent in the networking.
Should the need for double buffering be a very common occurrence, the file system can be implemented to support requests of the, "Give me block X, and I am going to be asking for block Y next so you might as well get prepared", variety. After providing block X the file system can go off and get block Y ready so that the next request of this type could be responded to swiftly. Implementing double buffering in the file system can be useful but does introduce extra complications. Double buffering can be implemented without the file system supporting it by the typical message passing solution of "Hire someone". A process can be used as a buffering process, sitting between the file system and the ultimate user of the file. It can easily operate in an appropriate mode to support double buffering.

The need to overlap execution of the sending process with preparation of the response to the message can, in general, be taken care of in a system which simply blocks until a response is provided.

**Block When Buffers Full**

Another scheme which has been used in many systems is blocking of the sending process only when all the buffers are full. The system provides a pool of buffers which can be used to hold the messages until the receiving process actually receives them. One general argument given for this choice is one of speed. In the classic producer consumer situation where the producer can produce faster than the consumer can consume, it is argued that the producer would terminate earlier, thus freeing up machine resources at an earlier time. This argument must be tempered by realizing that the buffer pool consumes resources itself and the early termination of the producer may actually result in a negative saving. Despite this, the scheme is worth looking at more closely.

To be useful there has to be a large number of buffers or else the situation closely resembles blocking until reception. When the number of buffers is small, previous arguments indicate that, in general, blocking until response is to be preferred. As argued previously, if buffering is beneficial more processes can be introduced to provide buffering.

Another argument against blocking only when buffers are full is the same argument that was used against blocking until reception. A server process which needs to respond to a request should not block while responding. A system which blocks only when the buffers are all full, with a large enough buffer pool, will appear to provide this but it does not. A response sent when all the buffers are full will block the respondent. Should all the buffers
be full because the process being responded to has used them to send messages to the respondent the potential for deadlock exists. The situation need not be as simple as this. If all the processes which can free a buffer by reception are blocked trying to send a message, a deadlock will happen. The chance that this will happen is incredibly small, and that is really the problem. Because only the most unusual situation would cause deadlock, the chance that it would be detected during testing is even less than the chance that it would occur. As a concrete example, it is not desirable to have a missile defence system that operates correctly except if the enemy launches 1,234 missiles, 842 of which release 16 warheads, at a point in time when other tasks have caused a specific situation. This would definitely not be found in the limited testing which is possible. It would be far more preferable to have a missile defense system that deadlocks if the enemy launches any number of missiles, with any number of warheads, at any time. That should be caught during testing, and fixed. For a system that blocks only if all buffers are full, one way to guarantee that the deadlock would never happen would be to make sure that no process uses more than one buffer at a time, thus indirectly implementing a block until reception system.

It is interesting to look at actual systems which use message passing and determine what messaging can make use of sends blocking only when buffers are full. In general the handling of the messages by the receiving process must never result in an error condition. If used, for example, to hold the output of some process which is destined for a disk file, the disk should never be filled otherwise some of the messages will be “lost”. When some status response is needed, all that has been gained by allowing blocking when full is to introduce extra complexity into the implementation of the primitives, and into their use.

Based on these arguments, blocking only when buffers are full can be dismissed. Anything that is gained by the use of the buffers can be provided, admittedly at some expense, by the introduction of buffering processes where useful.

There is a means of using a scheme that blocks only when all buffers are full which appears to guarantee that processes providing responses will never block while providing that response [Fitzgerald 86]. An indirect mailbox style scheme with a special response buffer allocated to every mailbox, where any freed buffer is first used to replace any used response buffers, implies that a sending process cannot use up all the buffers and thus deny the respondent a buffer. On the surface this seems to solve the problems inherent in a block when full scheme but all it has done is make them enormously less likely. Consider a situation with two processes A and B, one mailbox M, and four buffers B1, B2, B3, and
B4. One of these buffers, say B1 is the special buffer associated with M. If A attempts to send more than three messages before B receives any, A will block. When B wishes to respond it would have B1 available as the special response buffer, and another buffer would become the special response buffer. This would occur immediately if there was a free buffer, or upon the next reception that B does if all were full. If B should hold back its responses until it had received four messages there would be a problem. B1 would be used for the first response, and there would be no free buffers if A had already refilled B2 to B4. Thus there would be no buffer to hold the next response of B. In such a situation the response primitives either would block the respondent until a buffer became free (never), or would refuse the response. Blocking B would result in deadlock. If this were not the chosen solution, what should B do when informed that the response was refused? The situation just described is easily handled if this was the true situation. The problem as stated was poorly handled. If B was a process which received messages from processes which wished to wait for a specific situation (midnight), and the buffers were all used by other processes, and midnight is detected, B would attempt to respond to a potentially large number of processes. Should it occur that these other processes are blocked and cannot receive the responses, deadlock will occur. As mentioned, the probability of this happening is almost zero. Still it is an avoidable risk, that can be avoided by blocking until reception or response as needed.

Never Block

The final possibility is that a sending process will never block. Never blocking the sending process implies that no buffers are used, and the sending process polls until the receiving process attempts to receive, or buffers are used and if all buffers are full the sending process is informed that the message was not accepted. What should the sending process do if a message is refused? This question is applicable in either case.

Either the sending process gives up, or it tries again. With no buffering, giving up on the first failure is obviously not reasonable. If buffers are used and it just happened that all buffers were full at that point in time, giving up on the first failure is again obviously not reasonable. Should the sending process have a finite limit on the number of attempts or should it try forever? With infinite attempts as the chosen solution, the machine will not deadlock in the classic definition, but the difference in results will be indistinguishable. The machine will appear to be very busy rather than very idle.
If there is a finite limit to the number of attempts, the message being sent will be lost. If it does not matter if the message is lost, why was it being sent in the first place? It seems much more efficient to not send the message and assume that it would have been lost. A lost message is serious but not as serious as a lost response. Should a process be in charge of printing spooled files, and should its request for the next block of the file it is printing be received by the file system, but the response to that message lost, it will never print the block it is waiting for, and the file system is never going to give the block to it again. Eventually some person will notice a quiet printer, and a disk full of spooled files and will probably do something about it. The serious question is, “Why get into this situation anyway?”

Much has been made of deadlocks in the previous arguments. Something must be said about blocking until response and deadlocks. Blocking until response does not guarantee that deadlocks cannot happen. Constructing deadlock situations in a system with a block until response scheme is quite easy. With other schemes constructing a deadlock situation is very difficult. This, while seeming to be a weakness in a block until response scheme is actually a strength. If a deadlock situation is possible, the block until response scheme almost guarantees that it will happen, and at the first opportunity. It does not avoid deadlocks, it embraces them. Consider a message passing operating system as a car. All cars can have the brakes fail. A block until response car will guarantee that the brakes will fail when the car is initially driven off the assembly line, if they are ever going to fail, and long before the car is sold.

Given that deadlocks are so easy to create, it is interesting also that they are very easy to avoid, provided that the situation which can give rise to the deadlock is detected. The circular wait aspect of deadlocks is the aspect of interest. Consider a process as a node in a directed graph. If process A sends to process B, there is a directed edge from node A to node B. If this is an acyclic graph, there is no deadlock in the communications. If there is a cycle there are two possible ways to deal with this. Either an argument has to be constructed attempting to show that due to timing considerations such a deadlock is not possible, or the cycle can be broken.

Breaking the cycle is simple. If process A sends to process B and this is part of the cycle, introduce another process C which sends to A asking for a message to carry to B, and then sends to B announcing that the message is really from A. The cycle is thus broken, and the deadlock avoided.
This deadlock possibility can be detected before implementation is started. It can be detected as soon as the communication paths between the processes have been defined, and even before what is being communicated is fully specified. Algorithms to detect cycles in graphs are well known but using one of them is not even necessary. The message passing system itself can be used to detect loops. Consider two programs. One is a simple program which will be used by multiple processes, each simulating a node in the graph. The other program will be used by a process which creates the links in the graph. The creation process reads a definition of which process sends to which process; creates, at a higher priority than itself, the processes which will simulate the nodes; informs them of which processes to send to and which processes to receive from; and then tells each to start running. Each of the node processes will send to those processes it is to send to, and then receive from those processes it is to receive from. When it has done this it terminates. Being of lower priority than the node processes, the creator process gets to resume execution because all processes it created have either terminated or are blocked. If they all terminated, there is no cycle. If some still exist, a simple loop, checking the status of each process created, will indicate which processes are involved in a cycle and can indicate what the cycle is. This is of more benefit than an algorithm which finds cycles in graphs. Not only the processes directly involved in the cycle will be left, but processes which depend on them will still be in existence. The cycle is identified, and some of the effects of the cycle can be seen.

The final conclusion which can be reached is that the preferred scheme is blocking the sending process until a response, if the message was a question, or blocking it until reception if the message was a statement. Messages which are statements can be considered to have been responded to implicitly when they are received. Blocking the sending process in this scheme is simple conceptually, can be guaranteed not to disguise deadlocks, and is easy to implement correctly. While implementation details tend to be dismissed when designing a system they can greatly affect the correctness of the final product. Simplicity enhances the chances that the final product matches the designed system.

Section 2.1.3.2 Direct or Indirect Communication

If one process is to send a message and another receive it, some means of identifying both the sending process and receiving process is needed. Most descriptions of direct communication are based on the assumption that the identification of a process is based on
some sort of source level specification, while indirect is not. If this were the actual case, most of the arguments against direct communication would be valid. Direct identification is thus assumed to be analogous to a person's name, while indirect identification is analogous to a person's address. If communication within a computer were on the personal level this would be a valid assumption. Computer communication is on a business rather than personal level. Despite the growing use of the obnoxious "user-friendly" term, computers do not descend to the level of personal communication. A person can have a name, a title, and an address. Direct communication is analogous to communication with identification by title, while indirect communication is analogous to communication with identification by address. The valid distinction is that direct communication specifies a particular process while indirect communication only specifies a unique object. This unique object, commonly termed a mailbox, is analogous to a real-world address.

With direct communication the links are implicit due to the use of a process specific identification. No explicit creation or manipulation of mailboxes is needed, or possible. Any process which knows the title of another process can communicate with the other process. This lack of explicit mailboxes, and hence the primitives to manipulate them, results in a simple scheme for communication identification. There is the problem of the propagation of the titles of processes which still has to be attacked.

With indirect communication the mailboxes are explicit. This implies that primitives must exist, and be used, to create and manipulate these mailboxes. In a system using this form of identification the message passing primitives are not used to pass information from one process to another, but rather are used to manipulate these mailboxes. There is no need to propagate the identifiers of the processes involved, but the need to propagate the identification of the mailboxes is analogous.

For discussion the assumed method of communication is direct, and from this basis comparisons can be made to determine if indirect communication is a better choice. Each process is identified by a unique number which is generated at process creation (not program writing) time. The number is used for all process identification purposes, not simply for message passing. The generation of the number will be discussed later, all that is needed at this point is to know that for any given number, that number can only refer to at most one process at any time. Of the complete set of possible numbers only an extremely small fraction identify existing processes. It is also true that the receiving process of a
message can receive specifically from one identified process, and can also discover the identification of any process which is trying to send to it.

One of the common arguments against direct communication is that the propagation of the names of processes is a problem. To be more specific, the problem stated is the propagation of the change of a process name. Given that the identification of a process is not tied with the source of the program that it executes, this is not a problem, since a process never changes its identification. If process A wishes to send a message to process B it still must know the identification of process B in some way. This is the true propagation problem, but is not unique to direct communication. In an indirect communication scheme an equivalent sentence to the previous can be constructed. If process A wishes to place a message in link B it still must know the identification of link B in some way. Since the problem is equivalent in either direct or indirect communication, one is not superior to the other in this regard.

The true difference between direct and indirect communication can be seen when it comes to handling interrupted communication operations. For communication there are three active agents. There is the process sending the message, the process receiving the message, and the system which provides the support so that the message can be sent. Two aspects of invalid communication are important. One is what the system must do to deal with such happenings, and the other is what the innocent process must do. The situation in general is that some communication has been proceeding between two processes. There must be some protocol to this communication. One of the two processes violates this protocol by terminating without notifying the other.

With direct communication the identification of a process belongs only to that process. Only that process can receive messages marked for delivery by that identification. When that process receives a message it knows the identification of the process which sent the message. The identification of a process remains valid as long as that process is in existence. Should process A attempt to send to process B, and B no longer exists, it is simple for process A to be informed that the SEND operation is not possible because process B no longer exists. Similarly if process B attempts to receive a message from process A and A no longer exists, it is simple to inform B of the situation. Detection of a communication which is not possible is detection of the non-existence of a process.

With indirect communication the handling of such a situation can be more complex. The problem is the detection of the invalidation of a link. There are basically three
approaches that the designer of the message passing system can take. The simplest approach is to do absolutely nothing. It is easy to implement, but may place too heavy a burden on programmers who use the system. Another approach is to associate a link with an owner process. This would give some support to the programmers without an excessive demand on the message passing implementation. A final approach is to completely manage a link. This is by far the most complex, and so will be looked at first.

To completely manage a link there must be three lists associated with each link. There is a list of processes which can place messages in it, a list of processes which can take messages out of it, and a list of processes which control it. The justification for these lists is obvious. If interrupted communication is to be detected these lists are required.

Consider the situation where there are two processes, a producer and a consumer, and a link between the two. Consider also that the three lists are not kept. The producer terminates without informing the consumer. The consumer will wait forever for another message, which will never come. There is no means for the system to inform the consumer, or even be aware that it should. A similar situation exists if the consumer terminates without informing the producer. If the system could know that there never will be any process capable of placing a message into a link, or removing a message from the link, the situation could be dealt with.

To have this knowledge the system must know the capabilities of every process with respect to this link. The information is only needed to deal with an early termination situation, but must be kept updated at all times. For every link a process has six potential capabilities. These capabilities are:

1/ the capability to add a message to the link
2/ the capability to remove a message from the link
3/ the capability to grant to any process the capability to add a message to the link
4/ the capability to grant to any process the capability to remove a message from the link
5/ the capability to grant to any process any capability to grant that it has
6/ the capability to relinquish any capability it has

To deal with early termination of a process the capabilities of the terminating process have to be revoked for all links. For a process which does not explicitly relinquish a capability the system has to implicitly relinquish it on termination.
If no process has the capability to add a message to the link, and no process has the capability to grant the capability to add a message to the link, the link is useless and effectively invalid. When there are no more messages in the link, and it is useless, it becomes invalid, and any process which attempts to remove a message from the link can be informed.

If no process has the capability to remove a message from the link, and no process has the capability to grant the capability to remove a message from the link, the link is useless and immediately invalid. Any process which attempts to place a message in the link can be informed.

To deal efficiently with a terminating process a list of links it could affect has to be kept. To deal with the invalidation of a link a list of processes that are affected has to be kept. This implies that in effect there is a sparse three dimensional Boolean matrix. One dimension is indexed by process, another by link and the third by capability.

This complexity has to exist to deal with the early termination of the producer or consumer, in order to provide the same benefits as a direct scheme.

Should the second approach of associating a link with an owner be used, the need for these lists can be reduced. The owner can be defined to be the only process capable of granting the permissions to add or remove messages from the link. This results in two lists, one for adders and one for removers. Two possible positions to take on the termination of the owner are possible.

The simplest position to take is that the termination of the owner of the link terminates the link. This is very similar to the situation with direct communication and so would not be of any advantage over direct communication.

The more complex approach is to terminate the link if there are no senders and the owner has terminated, or no receivers and the owner has terminated. The process X could own the link, grant adding capabilities to the producer, grant removing capabilities to the consumer, and then terminate. Should either the producer of consumer terminate, the link would automatically terminate. This again gives the same behaviour as the direct communication scheme. Should the producer or the consumer process be the owning process, the situation does become a little less intuitive. The rules for termination change to reflect this extra complication. If there are no adders, and the owner of the link has
terminated, or there are no adders and the owner of the link is attempting to remove a message from the link, the link can be terminated. If there are no removers, and the owner of the link has terminated, or there are no removers and the owner of the link is attempting to add a message to the link, the link can be terminated. These rules are very close to those needed for the complete management of the links. Indeed, it shows that the rules given for complete management glossed over a fine point. The definition of “... no process has the capability to grant the capability to ... a message ... the link, ...” must be changed to “... no process which is not blocked attempting to ... a message ... the link has the capability to grant the capability to ... a message ... the link, ...” which is rapidly becoming incomprehensible. Even in the owner case the rules are becoming complex enough to indicate that direct communication is to be preferred.

The simplest solution, the system does nothing implicitly with links, is very easy to implement and define to all users. These are very good points in its favour. The unfortunate effect however is that a process must not terminate without informing all processes that it is communicating with, that it is terminating. It would so complicate programs, that there is no need to look at this approach in depth. In general all programs would behave correctly, but attempting to prove that all programs in all situations terminate correctly is perhaps too difficult. It is the same as proving the programs correct.

Given this excessive complexity inherent in an indirect scheme there must be extremely powerful benefits to be gained over direct communication to make it worth using. There are three benefits of indirect communication which are not available in direct communication.

1/ Knowing the identification of a link does not grant any right to use it.
2/ Multiple processes can pick messages up from a single link.
3/ A process can pick up a message from any of a number of links.

In general these three advantages of indirect communication are neither used nor needed. Should they be, a direct communication scheme can be used to simulate them by the, as usual, creation of another process.

If a process exists which simulates a link, any process wishing to perform an operation on the link need only send a message to the link process. That link process can enforce any capability checking it requires.
A link simulation process may also allow multiple processes to remove messages from the link by accepting a removal request from more than one specific process.

The ability to remove a message from one of a set of links is more interesting. This either implies that a process can choose which link to remove a message from, or that it can indicate a set of links and the message passing system would remove a message from one of the set of links should a message be available in any of the set.

If only the need to choose which link to remove from is desired, the removing process need only send a remove request to the appropriate link simulation process.

The set of links situation is not as obvious. The simplest way to implement this is to have the remover process repeatedly poll all the link processes in which it is interested. While simple, it has the disadvantage of excessive processor utilization. Another approach would be to have yet another level of processes which would ask the link processes to respond to them if a message was available. When one was, the assistant process could inform the remover process. The remover process need only wait for one of the processes, assisting the links from that set, to inform it of a message. This can also be complicated by the existence of a set of processes attempting to remove messages from disjoint sets of links. Should this be the case, some overall controlling process would have to be created which could integrate all the requests into a manageable whole. As will be seen later when the prototypical processes are discussed, such a process tends to exist, although not necessarily for this specific situation.

The bottom line is that indirect communication has no added benefits over direct communication, but does complicate the message passing implementation if done correctly. From this it can be deduced that direct communication is the preferred scheme.

2.1.3.3 Receive Primitives

A message passing system needs primitives to send messages, and primitives to receive them. Continuing with the basis chosen, there are two RECEIVE primitives. There is a specific RECEIVE which receives only from the process named, and a general RECEIVE which will receive from any process which is sending to the process attempting to receive.
For a server to function it must be capable of receiving from any process which attempts to make use of its services. It must include not only all processes which existed when the server was created, but all processes which have been created after the server. The general RECEIVE which will accept any sending process is sufficient to support the existence of servers. It has been argued that this is the only receive primitive necessary in a message passing system. Provided that there is some way of determining the non-existence of a process, this is true. For the simple producer consumer situation previously discussed the detection of the early termination of the producer would not be sufficient to inform the consumer that no more messages would be produced. If it were using the general RECEIVE, the system would have no way of deducing that all senders to the consumer had ceased to exist. To deal with the problem of early termination with only the general RECEIVE there are two methods of attack. One is to complicate direct communication by adding the lists mentioned in the discussion of indirect communication. This, however, would introduce an excessive amount of complication to what should be a simple situation. A second approach is to implement a primitive which blocks until the termination of a process. This "vulture" primitive could be used by a supporting process to detect the early termination of the producer, and then inform the consumer process.

If a specific RECEIVE is also implemented, the producer consumer situation regains its simplicity. The consumer need only receive specifically from the producer. The system can easily detect the termination of the producer and hence the impossibility of the consumer ever receiving a message from the producer. The specific RECEIVE is thus the mirror of the specific SEND.

There may be a need for a "vulture" primitive in some situations. If a support process attempts to receive from the process which is to be monitored, it would block until either it received a message, or the monitored process terminated. The failure of the RECEIVE operation would serve as the indication that the monitored process had terminated.

Rather than providing two RECEIVE primitives, only the specific RECEIVE need exist, provided there is a means of obtaining the identification of a process attempting to send. To implement a general RECEIVE in a server this second primitive is used to acquire the name of a sending process, then the specific RECEIVE is used to receive the message. To deal with inopportune termination of a process these two primitives have to be used in a loop so that, should the sending process be terminated after the server has been informed of the identification of the sending process and before the specific RECEIVE
was done, the server could wait for a message that was finally received. It is obvious that if a general receive were directly implemented, it could be more efficiently handled than by simulation with two primitives.

The cost of implementing a second general receive primitive is minor. So that the order of the sends can be maintained an ordered list of processes which are blocked attempting to send has to be maintained. A general RECEIVE need only find the first process in the list of those sending to the receiving process. It may even be advantageous to implement this single list as a set of lists, one for each process being sent to. This implies that a general RECEIVE need only check to see if this SEND blocked list is not empty. In any case, the primitive which would determine the name of the process which was attempting to send would have to go through the same sort of algorithm as would the general RECEIVE. The only major difference would be that the general RECEIVE would be able to complete the reception without further actions on the part of the process making the receive request.

Some [Matelan 85] argue that there is a need for a primitive which can be used by a receiving process to enquire if there is a process attempting to send to it rather than a primitive which blocks the receiving process until there is a sending process. There is a strong reason for not implementing this enquiry primitive. There is a large body of programmers who are conceptually wed to a polling approach, and never felt comfortable with interrupts. The advent of some new personal machines which provide what are called “event driven” environments have also led to this situation. These environments typically provide a “null event” with which the programmer is placated. Thus the programmer is placed into a direct polling environment. The primitive to enquire about the existence of a sending process would induce this body of programmers to assume a polling attitude when writing programs. It is unfortunate, but generally true, that to programmers the existence of a “feature” implies that it must be used. It has been previously shown how asynchronous termination could be used to deal effectively with a flood fill operation in a graphics application. Provision of a polling primitive would most likely lead to a situation where the flood fill process would periodically poll to see if a process was attempting to inform it that the flood fill should be terminated. Apart from consuming processing resources unnecessarily, this approach would also complicate the implementation of the flood fill.

Another aspect of the receive primitives apart from specific and general receives is something which can be termed request code screening. In some situations it would be
convenient if the requests received were restricted to specific subsets of all requests possible. For example, in the case of a process which is acting as a buffer between a set of producers and a set of consumers, request code screening can significantly simplify the program of the buffer process. Consider the situation where there are no messages in the buffers. In such a situation the buffer process must queue any consumer requests it receives until such time as a producer sends a request. A similar situation exists with the producers and consumers reversing roles when the buffers are all full. If only producers sent requests when the buffers were empty, consumers when the buffers were all full, and either when only some of the buffers were in use, the program of the buffer process would be greatly simplified. If the message passing system supported request code screening, such a "desirable" sequencing of events could be assumed. With all buffers empty the buffer process could arrange that only requests providing data to fill buffers would appear. With all buffers full the buffer process could arrange that only requests emptying buffers would appear. In the intermediate state it would accept either type of request.

The benefits of request code screening are high but the costs must also be considered. The implementation of ignoring requests which are not of the appropriate types is easy and inexpensive. The complex and expensive tasks are determining which types each request can be classified as, and how to encode this information. For a single server there is a clear set of requests which are acceptable. These are easily grouped into various types. Many servers can respond to almost the same set of requests. For simplicity and comprehensibility these requests need to have the same encoding for all servers. For example, to a client process, asking for the next “n” bytes from a file should be the same as asking for the next “n” bytes from a serial line. For the file system server a read request would not be classified as any different from any other request such as a write request. A serial line server might desire to have all read requests ignored until there was actual data available from the serial lines, while still accepting write requests. Thus, for each server which would make use of request code screening, each request would have to be assigned a type. This could imply a unique mapping of requests to types for each server. This mapping has to be done by the message passing system without active participation by the individual servers. The management of such a potentially large amount of specific information is a cost which may be too high.

If the classification of requests could be made independent of the individual server, there would be more reason to still consider request code screening. A server making no demands on the services of request code screening needs to indicate that all types of request
are acceptable. The file system server would accept all requests, while the serial line server would use restrictions. This seems reasonable, however there is one slight difficulty to unification.

When a new server is being designed the functional mapping of requests to types has already been defined to a great extent by the servers which previously existed. Should the new server require a different grouping, new types of requests have to be defined. This obviously implies that a single request can be classified as a set of types. There are now a set of acceptable request types specified by the server using request code screening, and a set of types that a given request can be classified as. The implementation of request code screening is reduced to checking if the intersection of two specific sets is not null. These two sets must be implemented in a manner which places no apparent bound on the potential size of the sets. The costs of request code screening are mounting and if some other solution can be found which provides the same benefits, it should be considered.

As usual, the addition of intermediate processes can be beneficial. Consider the serial line server process. One major reason it would be designed to use request code screening is that the number of processes which could be requesting input from the serial line, while no input was available, is potentially large. The server would have to record at least the information of which process wanted input from the serial line. It may be assumed that receiving from some specific process rather than from any process when there is no input available is not acceptable. In this situation the insertion of a single process between the serial line server and all processes requesting input would solve the indeterminate queuing problem. All processes requesting input from the serial line server would send requests to the intermediate process rather than the serial line server directly. The intermediate process would forward the message to the serial line server. The length of the queue which the server has to maintain for input requests which cannot be satisfied is now a maximum length of one. The benefits of request code screening have been provided, without request code screening, at the cost of an extra process.

Another point against request code screening is that to provide it, the implementation of the message passing primitives must be able to identify the request of the message. Without request code screening the implementation of the primitives need have no knowledge whatever about the message, only to which process it is to be given, and whether or not a reply is required.
While request code screening is conceptually desirable, it is not necessary. The simplicity of no request code screening implies that only a specific \texttt{RECEIVE} and a general \texttt{RECEIVE} with no screening should be implemented.

2.1.3.4 Responses

Previous discussions have lead to a situation where a separate response primitive is required. Sends block until either an implicit response is given when the message is received, or until an explicit response is provided. The process providing the response will never block.

The exact implementation of the primitives will determine how the implicit and explicit response messages are separated. If it is possible for the message to imply an implicit response, while to the server it implies an explicit response, a potential problem can arise. It is best if servers are protected as much as possible. Providing a reply to a process which is not blocked waiting for one should result in a status indicating this fact, but no adverse effects on the process providing the response. Since a process can be terminated while blocked for a response, responding to a process which no longer exists must also be safe, and only result in a status indication. This termination can result in the situation where a response is provided to a process which exists but is not waiting for a response. This is the result of the finite size of the process identifier. Being finite, there is a potential situation where the response has been delayed long enough that the process identifier has been given to another process which was created after the termination of the original owner of that process identifier. While true that given a reasonably long period of time most probably has to pass before the process identifier will be reused, this is all the more reason for allowing the response to be rejected with no adverse effects. Having the spacecraft system terminate with such a situation after 300 years of operation seems cruel.

One question worth discussing is which process or processes can respond to a response blocked process. It is reasonable that the process which received the message should be able to respond to it. Is it reasonable to assume that other processes can as well? This should be looked at as a set of rules with increasing generality. The first and basic rule is that the process which received the message can respond to the message.

A second rule to consider is that any process on which the sending process is ultimately blocked can provide the response. A process A can respond to any process which
is blocked waiting for a response from process A. Further, a process A can respond to any process B which is blocked waiting for a response from a process C, if process A can respond to process C. The only benefit from this extension is that intermediate processes would not be directly involved in conveying the response to its ultimate destination. No intermediate processes could be responded to before the ultimate process was responded to. An undesirable aspect is that an intermediate process would have to "know" whether the response was responded to by the final receiving process or not. It is simpler to let the response "percolate" back through the chain of intermediate processes, even if the intermediate processes need not become involved with the contents of the response that they are passing.

2.1.3.5 Message Type

A message can consist of up to four sections. The first section is a fixed number of cells of a pre-defined type. The second section is a fixed number of cells of a user defined type. The third section is a variable number of cells of a predetermined type. The fourth section is a variable number of cells of a user defined type. Given that a cell can be a heterogeneous record this general message is sufficient for all communication needs. Any specific system will define a message as some subset of these four sections.

The exact form of the messages allowed, and whether or not the message must be copied are interrelated. If the message is of some system known size, it is easily seen that memory management hardware could possibly be used to support "swapping" message segments rather than copying them. This would remove some of the objections to large messages since the size of the message would not affect the message passing speed. It would mean that large messages could be used, thus providing enough fields in the message to satisfy most communication needs. This in turn could remove the need to support user defined types of messages. If it is assumed that messages need not be copied the discussion can start with a message being a fixed number of cells of a pre-defined type.

Given that the message is of a fixed size it is still possible to support both homogeneous and heterogeneous messages. The implementation of the message passing primitives need not be, in general, concerned with the internal representation of the message. With the message area limited to a region defined by memory management hardware, so that copying is not necessary, the primitives only manipulate the description of the area of the message and not the message itself.
When the system is extended to support message passing between machines connected to a network there are other factors which have to be considered. For a network of homogeneous machines there is no need to change any decisions which are made based on a single machine. Heterogeneous machines do have an impact. The basic problem is that the representation of a single type of object can vary between machines. For example, the order of the bytes in a 32-bit object varies between different machines. This minor problem is easily overcome by defining the ordering of bytes on the networking medium. The major problem is deciding when a sequence of bytes in a message represent a 32-bit number. For example, consider a message consisting of four bytes being sent from a “little-endian” processor to a “big-endian” processor. Assume that the bytes are labeled 1, 2, 3 and 4. If the four bytes represent four characters the big-endian machine would store them as 1234. If the four bytes represent two 16-bit numbers it would store them as 2143. If the four bytes represent a single 32-bit number it would store them as 4321. The problem is that when networking is considered the type of each component of the message must be known before it can be correctly transmitted on a network. A system defined message satisfies this requirement.

Allowing user defined types of message is appealing. It provides for situations which the designers of the system did not consider. It does require a user defined description of the message should the message be transmitted across a network. Since the sending process need not, and possibly will not, know if the message is crossing a network, then for every message sent, a definition of the internal structure has to be provided. A description can be provided at compilation time for fixed structures but would have to be constructed at run time for variant records such as are provided by various programming languages. A sufficiently large system defined message with a reasonable number of distinct basic types of fields is more desirable from a complexity argument.

The size of the message is important when networking is concerned. While messages need not be copied within a single machine, they must be copied to transmit them over a network. The difference between a message of length 120 bytes and a message of 130 bytes is insignificant when a single machine is considered. If the network packet size is 128 bytes the difference can be very noticeable. If some means were available to indicate which fields contained information and which fields were not important, only the useful fields need be transferred over the network and the 130 byte message may well fit into one 128 byte packet.
This intricate situation may best be avoided for now. It seems intuitively obvious that message passing extends to inter-machine communication. What is not intuitively obvious is that extending message passing so that there is no distinction between a message which is local to a machine and one which is crossing a network, is not necessarily a reasonable thing to do. Questions of user validation and network partition need to be addressed as soon as a network is used. This is a large area which cannot be discussed here. It is sufficient to state that processes which communicate across a network may very well be using programs which were written for that exact situation. As such, these programs could use a different form of message passing which more directly matches the communication medium. If that is the case, the difficulties raised here can be avoided by shifting the responsibility to the programmers of such programs.

One aspect which has been ignored until now is the provision for a variable length part of a message. Where would a variable length message be useful? A moment’s reflection soon identifies file operations as a potential area. A process dealing with a file needs to read and write some area of the file. The exact size of the area may only be known at execution time.

It is appealing to consider that the message area defined by the memory management hardware can be of a variable size. If either paging or segmentation hardware is used it is reasonably easy to support variable sized message areas. Two points argue against this however.

The data read from or written to a file is not generally useful in the message area. It will have to be copied into the message and sent, and the receiving process will have to copy it out of the message area on reception. This is obvious if a process which merges two sorted files is considered. Two input buffers are emptied to fill one output buffer.

Another problem with attempting to associate the variable portion with the message is that at any point in time the variable portion of the message may be the wrong size for the required data. This means that the message areas must not only be of various sizes, but must dynamically vary in size. Not only must the message area grow in size as needed, it must be reduced in size as needed. Consider that if one process uses a variable size of one million bytes at one point in time. The area is given to another process to use as a message area, which would give it to another, and so on. There is no guarantee that it would be the area used to return the response to the request initially sent. Some process somewhere now
has a message with a variable size area of one million bytes which it does not need. If enough messages are sent of this large size, eventually all processes will have huge message areas and the machine will have effectively shrunk in size. Each SEND operation must also imply a potential dynamic memory operation. With segmentation, it can mean compacting memory. A paging system need not involve such a great expense, but even so there is some expense involved. Message passing operations should be as swift as possible so it would be convenient if there were no variable sized portion of a message. In such a case the message area would remain of a fixed size. No dynamic memory operations would ever be required for a message operation.

A simple solution is to treat the fixed and variable sized portions of a message as two distinct objects. Many communications will not involve a variable part and so that part is best left to alternate means of transmission. By providing a separate means of moving variable amounts of data, massive amounts of data can easily be handled. These data can be moved just once, from the initial location to the final location, no matter how many intermediate processes are involved between the originator of the data and the recipient. This massive data movement must now be considered.

2.1.3.6 Massive Data Movement

There are times when data do not fit nicely into a message. The most obvious situations occur with file operations. When a process requests that a piece of a file be read, the size of that piece is determined by the process making the request. The response, if all data were moved by messages, would have to be of a variable size, while the request would be of a small fixed size. When writing to a file it is the request which is of a variable size, and the response which is of a small fixed size. Many of the intermediate processes between the originator of the message and the file system would have the variable parts of the message pass through their address spaces, though they have no reason to either read or modify it.

Rather than passing all data in messages an alternative method can be used. The situation is analogous to furniture moving in the real world. When a person moves from one city to another and uses a moving company for the task of moving the furniture, the furniture does not flow in the same path as the orders do. The furniture stays at the source until picked up and moved to the destination. It does not go through the office of the receptionist of the moving company, or the office of the dispatcher, or the office of the receiver, or the office of the accountant who records the payment. Nor does the person,
who is having the furniture moved, pack it and take it to the moving company when requesting that the furniture be moved. Should another person be receiving the furniture at the destination a letter of notification can arrive without the furniture attached. The message originally sent indicates where the furniture is, what it is, and what to do with it, but does not include the furniture. This is the model proposed.

Some restrictions on data movement are necessary. If any process can read and write any location in the address space of any process, all arguments of correctness are futile. Furniture should be removed or delivered only if the owner involved has stated that this is the desired operation. This fits nicely with the concept of blocking until there is a response. If a read from a file is requested, the process which is to receive the data can state at the time of the request the area which is to receive the data. Similarly, when writing to a file, the area containing the data can be stipulated at the time the request is sent. Data can only be moved into or out of the address space of a process when it is blocked waiting for a response.

The process exposing an area should have some control over exactly what is exposed, and how. It would be disturbing to be expecting a chair to be delivered, only to discover that the television had been removed. The exposing process should specify where the area starts, the length of the area, and the permissions given. The permissions indicate whether the variable part is being "sent" with the message, returned with the response, or both.

It is worth considering at this point whether or not more than one area need be exposed at any given time. In all the years of operation of both Thoth and Port only one situation ever was noticed which required two areas. If a file is to be moved within the file system, two file path names, the source and the destination, have to be provided. If moving a file is to be considered an atomic operation, both path names have to be provided in one request. Having only one area exposed means that both path names would have to be placed in one area. While potentially inconvenient, the simplicity of one exposed area far outweighs the few times it is a restriction.

Another interesting point is that the receiver of the request need not be explicitly told about the area exposed. For example, should the file system be requested to read a block of some size, the permissions should allow the modification of the area written. A write request implies that the exposed area can be read. The size of the area has to be passed, as would be expected for a read or write in just about any system, but the address of the area does not. The massive data movement primitive only works with the exposed area so its address is implicit in the operation.
The movement of the data need not be performed in one operation. The implementation of the movement primitive would be simplified if this were the case but the usefulness of the primitive would be under question. If a process requests 2,387,562 bytes to be read from a file, the serving process would require a buffer 2,387,562 bytes in length to handle the request. This would be true even if the device used to store the file in question could only provide 1024 bytes at a time. It would be far better to allow the process using the movement primitive to specify an offset into the exposed area, and a length when either taking or giving data.

The situation can arise, since the original sender is in control of the exposed area, that the process which attempts to service the request may attempt to violate the restrictions to be enforced. It is possible for an ill-constructed process to expose 1024 bytes with no modification permitted, while asking the file system to give it 2048 bytes. As usual it is sufficient to inform the process using the movement primitive that the operation was not successful, since it is probably a server of some kind.

Section 2.2 The Hardware

Not only are there desirable features of the operating system to be discussed, but also of the hardware. One area worthy of discussion is memory management.

2.2.1 Memory Management

The first question to consider is whether memory management is necessary, or even desirable. All compiled references to memory locations could be generated as offsets from one of some set of “base” registers which could be fixed at the time the process starts. Machine registers used as “base” registers by the generated code from compilers would even allow multiple processes to share the same program. Given such a scenario it can be firmly stated that memory management is not necessary. The success of some current personal computers which have no memory management hardware also would tend to lend support to such a decision. One can also look back to the success of the 360 family of machines.

The desirability of a machine with no memory management hardware is more in question. Arguments presented previously against a shared memory model and for message passing are relevant here. Given a set of programs which were “perfect”, the processes
using those programs would also be "perfect". Thus there would be no need to have memory management hardware of any kind. Perfect programs form only a small proportion of all available programs. If this proportion was large, the situation would not change. One imperfect program would create at least one imperfect process. The execution of that imperfect process could damage the perfect programs and spread its "imperfection". Thus, even though memory management hardware may not be necessary, it is desirable. Some means of protecting one process from another localizes the damage perpetrated by a process using an imperfect program, to the address space of that process.

Allowing two processes to share the same program is possible in a system without memory management hardware. The introduction of memory management hardware, if it only provides one address space, can make this impossible, since both the program, and the data that it modifies, must fit within that one space. If two versions of the data exist, two versions of the program also must exist. The sharing of programs by multiple processes is generally assumed to be desirable but this too should be looked at in some detail.

For a large time sharing system, sharing is indeed useful, since more than one user may well be using the same program at any given time. For a personal machine further investigation is needed. A personal machine, while it may be a "single user" machine, need not be a "single use" machine. The fallacy of equating these two terms has unfortunately been prevalent. For machines and systems designed on this fallacy the level of frustration of the user tends to increase with the level of sophistication. On a single user machine the number of times there will be more than one "user" using the same program will tend to be less but need not be zero. The memory management hardware is not connected with whether programs are shared or not. Sharing programs means that the program should be protected from modification while in memory. This is the true concern of the memory management hardware. Whether the feature is used to support shared programs or not is the responsibility of the operating system.

The desirability of the provision, by the memory management hardware, of protection against modification of programs in memory has been adequately discussed previously. It can be accepted as having been shown.

Memory management hardware has some bearing on whether or not virtual memory is supported by the total system. It must supply some indication that an address which was used was acceptable or not, but that tends to be a function which can be considered apart from virtual memory itself. Virtual memory is based on one conceptual assumption. It is
assumed that a process will exhibit “locality of reference”; during some small period of time a process will access a small proportion of the total address space that it has. If the small proportion can be identified, only that proportion need be physically available to the process. Over time the exact small proportion may change but will remain as a small proportion. The granularity of the memory management hardware can affect the viability of a virtual memory system. If the memory management hardware works with pieces which are a minimum of one million units in size, the probable proportion of the program that is needed at any given time is going to be relatively large. If virtual memory is to be supported, the memory management hardware must use a small granularity, or else any benefits that virtual memory provides will be obviated by the negation of the assumption that just a small proportion of the total address space is accessed during a short period of time. Whether virtual memory is desirable or not is a separate question which should be covered first.

Virtual memory is a reasonable concept to support in a time sharing system. In such a system there is a perceived need to support many users at any given time, and to attempt to make it appear to each that the others do not exist. The processor is shared between the processes of all the users and it does give some indication that other users exist since the effective speed of the machine decreases with the number of active users. If no means is provided of letting more programs execute than will physically exist in memory, the existence of one other user who is using a large proportion of the machine memory would have a significant impact on all other users. Swapping programs between main memory and some backing store can make the machine appear to have much more memory than it really does, and can lessen this impact to a great extent. Loading a large program from a backing store into memory does take a considerable period of time but the far greater impact is that those programs currently in memory must first be moved to the backing store before the large program can be loaded. Thus to load a program of size X into memory approximately twice as much information must flow, X to save those programs in memory, and X to load the program in question. Not saving those parts which have not changed can reduce this to some extent, but not completely. Not only is time spent saving those programs which were swapped out of memory, but for them to run again they have to be swapped back in, possibly forcing other programs to swap out. If a small proportion of each program could be left in memory, and that proportion is all that is needed for continued execution, much of the overhead could be avoided. This is exactly the benefit that virtual memory provides.
The cost of virtual memory is not zero. As the small proportion changes for each program some parts have to be saved and others loaded. Rather than paying a large time penalty all at once to swap a program in or out, the penalty is spread over a longer period. Whether or not the total penalty is greater for virtual memory or not is influenced by the interaction of the exact programs being used, and the strategies for picking which parts to remove from memory, and how much to load at once. For any given virtual memory system one program can be created which will prove that the system has the worst design possible. Simultaneously, another program can be created which will prove that the system has the best design possible. In general, for a reasonable mix of programs virtual memory tends to have less total penalty than a swapping system would.

Virtual memory works if the assumption that any process needs only a small proportion of its total address space at any one time to continue execution is valid, provided one other assumption is true. That assumption is that programs and their associated data are large. If the programs and data are small, the proportion needed increases. In the worst case the size of the programs and data can be smaller than the granularity used to divide programs and data for support of virtual memory. In the Port message passing system a large percentage of the processes use programs and data that are very small. Some programs are as small as 64 bytes and some data segments under 128 bytes in length.

Virtual memory is useful because of one other assumption which is valid in a time sharing system. It is assumed that the total memory requirements are much greater than the total physical memory available. If a large physical address space is available, and can be populated by memory, it is not necessarily a valid assumption for a personal machine. The basic problem with a simple swapping system is that to move a process with a large program and its associated data takes a considerable amount of time. If there is more than one such process these processes tend to "fight" for memory and induce a very large overhead. A user of a personal machine may do more than one thing at a time, but usually tends to do one thing, and some other things. Attention is focused on one area at a time. In such an environment swapping may be preferable to virtual memory.

A final point worth making comes from observation. An early version of the Port system supported swapping. It was noticed that nothing ever swapped since the available memory was large enough that it was never completely used. A version of the system was created which did not support swapping. Being a message passing system where all actions were only taken on demand this meant that the only difference in execution was that checks
that something was swapped were no longer made. For example, when attempting to move a message from one process to another, the test to see if the destination process was swapped (which it never was) was no longer made. It made a minuscule saving in size, and a minimal saving in time was expected. If fact the saving in time was not even considered, since the change was made for the sake of size and simplicity. Surprisingly the speed difference was highly noticeable. Two identical machines sitting side by side listing a large file to the screen, one checking to see if something was swapped (and never finding that it was), and the other assuming that nothing was ever swapped, quickly diverged in time. The quality improvement in the system which was not testing was overwhelming. Port never swapped again. The basic overhead in a virtual memory system, even when it never actually does anything, may be just too high a cost although it can be as low as a few percent [Cheriton et al. 88].

One important aspect of memory management hardware is the time taken to change a logical address to a physical address. Since every memory reference must suffer this cost it should be as small as possible. There are three aspects to the total overhead. One is the time taken to find what the base to add to the offset is, a second the time taken to compute the physical address from the base and offset, and a third the time taken to check the validity of the logical address. These three aspects are not completely separate.

If a paging model is chosen, the checking for validity is free since the time taken to discover the base value for the page covers the time taken to detect that the page is valid. The computation of the base plus offset covers the check for a valid access mode. If a segmentation model is chosen the check for a valid offset within the segment must be made, as well as checking that the segment is valid, and that the access mode is acceptable. Checking for a valid offset is expensive since it involves an arithmetic comparison, however this is also free since it can be overlapped with the physical address calculation since that calculation also involves an arithmetic operation. Validity checking can thus be ignored as far as time taken to convert logical to physical addresses is concerned.

A paging model computes the sum of the base and offset in a very inexpensive manner. Because all offsets with a page are valid, and the base address of a page is always a multiple of the page size, the sum does not require an arithmetic operation since the two values have no significant bits in common and can be implemented by routing selected bits from the offset and base to the final result. If a segmentation model is chosen, the sum of the base and offset is more costly. Because there can be an overlap in significant bits a true
addition is needed to compute the physical from logical address. As fast as addition hardware can be made, it will never be faster than a wire. A paging model definitely wins in this comparison.

To operate successfully in an extremely large address space yet still not waste too much memory due to internal fragmentation a paging model needs a very large number of pages. A segmentation model on the other hand can support a large logical address space, while still supporting both small and large segments with a number of segments which is independent of the actual size of the logical address space. To support a 32-bit address with a reasonable size of page it is easy to imagine that pages of size 8,192 are needed. Such a potentially large number of pages implies that either a large penalty must be paid to load and store these page numbers to high speed memory whenever a process is given control of a processor, or the overhead has to be spread over time by loading the values on demand. Certain schemes have been proposed to reduce this overhead [Thakkar 86]. A segmentation model for a 32-bit address can make do with many fewer segments, implying that the time overhead can be reduced. If the number of segments is small this overhead is unimportant. Segmentation can be better than paging when considering the time taken to obtain the base address from the logical address.

Both segmentation and paging have good aspects and bad aspects. The desirable situation would be to have the swift base address access possible with a small number of segments, and the swift computation of physical address from base and offset possible with a paging model.

Another aspect worth consideration is the amount of memory needed to store a copy of the information needed by the memory management hardware to support a single process. Having a few hundred processes in existence at any one time is not unreasonable. If there are 200 processes, and each requires 1024 pieces of information, 204,800 pieces of information must be stored. Use can be made of a goodly portion of a million bytes of memory. If a paging model is chosen, virtual memory may very well be necessary due to the expense entailed in keeping track of where each process is stored. If very large pages are used so that the maximal number of pages can be decreased the situation gets no better. The page tables stored would be smaller, but the pages needed to store them would be larger. If a large number of processes are to be supported, a small amount of information should be stored about each, implying a segmentation model with a small number of segments.
2.2.2 Multiple Processors

There are times when more than one process is capable of execution. This is definitely true in a time sharing system, and also in a message based personal machine. The computation speed-up possible from even two processors is great. The example given previously which had a 20% speed-up during compilation when using a slow network shows that more than one processor can be advantageous. Some number of processors is desirable and that number should definitely be greater than one. Increasing the number too much leads to a decrease in effective processor speed due to contention for the bus.

Ideally, if there were no problems with bus contention there should be one processor for each process. Processor allocation would be simplified, as there would be no need to interrupt one process to allow another to execute. It would be a terrible waste of processing resources since, in general, most processes are not capable of execution at any point in time.

Apart from the number of processors another important aspect is homogeneity. If all processors are identical the choice of which to use for a specific process is simplified. This would be a valid position if, indeed, all processes are identical. A process performing three dimensional graphics is going to use floating point operations more intensively than one which is formatting text for a book. Apart from what the process is doing, the language in which it is written is also important. A processor which can efficiently execute programs written in LISP will be different from one which can efficiently execute programs written in COBOL. One general processor type can be used for all languages, however the match between provided instructions and conceptually required instructions will not be exact. Systems and support processes, and a large number of application processes can function reasonably well using a single processor type, and that type should form a large percentage of the total number of processors, but heterogeneous processors should be supported.

2.2.3 Cache Handling

Given a multiple processor architecture there is a need for cache memory if for no other reason than to lessen the use of the common bus [Briggs 83]. The existence of a cache must in no way change the results of computation.

Accessing a memory location which is held within the cache should be faster than accessing the memory location in common memory due to the elimination of the need to use
the common bus. Accessing a location which is not held in the cache takes longer due to the overhead induced by the cache itself. The location may not be held within the cache if it has never been accessed before, or if it was forced out of the cache either because the space used for it was needed to hold another location, or because of a need to force the cache to "forget" what it was holding. The first two reasons for not having the location held in the cache are always valid. The forced "spilling" of cache contents must be under some control. "Spilling" everything because one location changed is not an acceptable solution.

If the cache were conceptually a set of caches, each could be treated individually. The forced "spilling" of the cache contents could be restricted to that small section which was affected by the cause for the "spilling". These conceptual caches should be of variable size. Providing one fixed size for each section of the cache is simple, but would not clearly reflect the access behaviour of a process. All the locations in one section may be accessed and only a few of another. The sectioning of the cache must not be done by providing multiple caches, but by having the cache treat sections of the address space in differing manners.

Locations accessed by some means other than through the particular cache, can change without the cache being aware of the change. If this set of locations were never accessed through the particular cache, no need to "spill" the cache would exist. Certain areas may simply not be cached.

Some locations are, by definition, accessed by multiple processes, be they software or hardware processes. The most obvious are device communication areas. It is totally unreasonable to cache these sorts of locations because their very volatility would force cache "spilling". Caching them would provide no benefit, it would just slow access to those locations. It should be possible for certain areas to be marked as not cached.

In conclusion, cached locations should not be "spilled" unless necessary. By the use of a set of logical caches, this can be approximated. Not caching those areas which are, by definition, not to be cached is simple. Restricting "spilling" of a cache location in a location by location manner may be too expensive, but if the sections of cache are properly assigned this strict requirement may be loosened without too much overhead.

### 2.2.4 Co-processor Integration

One of the common means used to augment a processor is by the introduction of co-processors to provide instructions which can be simulated in software, but are more
effectively performed in hardware. This is quite often seen in the provision of floating point operations.

If the processor is to be used for scientific operations, it can include instructions for the manipulation of floating point values. If the processor is to be used for commercial operations by programs written in COBOL, instructions which manipulate blocks of memory and format values for printing can be included. If list processing applications are common, instructions to manipulate stacks and queues can be included. This full complement of instructions can be provided in two basic ways. A single processor can be built which supports all the instructions required, or a simple processor that provides the basic instruction set can be built and co-processors can be designed which provide a set of instructions that the basic processor does not.

Building one processor which supports all the instructions has implications. First, that processor is going to be physically large since the circuitry for all the instructions is going to be large. Second, the realization of the processor is going to be complex. Unless the implementation of each of the sets of instructions are kept disjoint, making it even larger, there is going to be a high integration which can lead to complexity. Third, the cost of every processor is going to be greater than the cost of a basic processor. Apart from simply recovering the extra development and design costs, the larger chip area implies that any flaws in the chip substrate will affect a larger area since one flaw will damage a larger surface area. It will decrease the yield and increase the manufacturing costs. Fourth, the extra size and complexity of the integrated chip may well decrease the speed of execution of common instructions, due to simple propagation delays with longer signal paths, as well as other more complex reasons.

Building a basic processor, and a set of co-processors has implications. First, each chip in the set will be smaller than the integrated chip. The total area used will be greater than for the integrated chip but is of little concern since all the co-processors will not be used with every basic processor. Second, each chip will be simpler than the integrated chip. For the same reasons that a collection of communicating processes is simpler than an amorphous integrated process in software, a collection of communicating chips is simpler than an amorphous chip. Third, the cost of each chip will be lower than the integrated chip. The development costs for each will be less than those for the integrated chip, and by virtue of the smaller size of each, the manufacturing yield will be higher. The total development costs need not be higher. The complexity of an integrated chip may well push the
development costs beyond the sum of the development costs for the set of chips. Fourth, the need for chip to chip communication when co-processors are used may result in the execution time, for the instructions which they provide, being greater than for an integrated chip.

Apart from technical reasons for choosing one method over the other, there are also marketing reasons. A customer may prefer to buy a basic machine at a low cost and add co-processors as desired. A basic machine will provide software simulation of the instructions not available and the speed of those simulations, while not as good as the hardware speed, may be adequate for the customer's applications. Should the extra speed of the hardware be needed in the future, an upgrade path to enhance performance is available, preferred by the customer for cash flow reasons, and by marketing since it can increase the size of the customer base.

Using a basic processor to provide a minimal machine, and adding co-processors as needed to increase performance is the desired path. The means of integration of the co-processors with the basic processor is worth consideration. There are two methods of co-processor integration, visible and invisible.

A visible integration means that the instructions generated by a compiler for a given program, when compiled for a machine without a co-processor and for a machine with a co-processor, will differ. When compiled for a machine without a co-processor, either subroutine calls or in-line code would be generated to support the instructions not available. When compiled for a machine with a co-processor, the program would have instructions generated which deal with that co-processor. The integration can result in more efficient code for both cases. One disadvantage of visible integration is that when a co-processor is added, all programs which can make use of that co-processor should be re-compiled. This is a minor annoyance. Considered from a software developers view this is not minor. For each of the products that the software developer sells, there have to be multiple versions available. As well as the basic version there is the version which uses co-processor A, the version which uses co-processor B, the version which uses co-processor A and co-processor B, ... and can soon become a management problem.

The major problem becomes evident when a co-processor becomes faulty. The co-processor cannot be removed until all programs which use it are re-compiled, or other versions purchased if the program was bought. If the co-processor fails in a major way, so
do all programs which use it. This can be very serious if the compilers needed to re-compile the programs also use the co-processor.

Another aspect worth considering is that of the existence of multiple processors. If the compiled program "knows" that a co-processor exists, either that program is restricted to the set of processors on which it can be used, or the co-processor has to be purchased for every processor. Either alternative is possible but neither may be acceptable. Overall a visible integration is not desirable.

An invisible integration of a co-processor can be affected in two basic ways. The basic processor can either appear to support the instructions which the co-processor supports, or the co-processor can be accessed by subroutine calls. In either case the use of the co-processor is not going to be as efficient as in the visible case since the code of the program has to cater for either a hardware or software implementation. The benefits of invisible integration are great. Only one version of the program need be created. It will make use of a co-processor if one exists, or use simulating software if not. The effect of adding a co-processor is that things run faster. Nothing else need be changed. Should a co-processor fail, it can be removed and the effect is that things run slower. If not all processors in a multiple processor machine have a co-processor, certain programs will be executed more swiftly on some processors than others. If the operating system is aware of which processors have what co-processors, and which programs would "like" to use what co-processors, it could make a reasonable choice as to where to have a given program executed. If no processor is available with the preferred co-processors, the process using the program can still be given another processor to use.

It has been common to have certain instructions, in the instruction set of the basic processor, access the co-processor if one is present, or cause some form of software trap or indirect subroutine call if not. The use of the co-processor, if available, is efficiently supported. In the case when the co-processor has to be simulated, the subroutine call is not as efficient as in the visible case, but the overhead compared to the time taken to simulate the instruction is small. One undesirable feature of this solution is that the basic processor has to be aware that a co-processor does exist. For each co-processor that can be added, the basic processor has to be capable of deducing its existence. The processor either has to "know" or it has to "ask". More fundamental a problem than having the basic processor become aware of the existence of the co-processors, is the need to design the basic
processor with the knowledge that co-processors will exist. This adds complexity, and complexity which can be avoided should be avoided.

The other approach to invisible integration is to hide all co-processor operations in subroutines. It is a method which is quite often chosen when the hardware provides only visible integration but the benefits of invisible integration are considered more important than the benefits of visible integration. Two sets of subroutines are created. One set consists of software simulations of the instructions of the co-processor. The other set interfaces to the co-processor itself. The appropriate set of subroutines can be linked with the program when it is loaded. The program need not be compiled with the knowledge of which co-processors are available. There is extra benefit over the other method of invisible integration. If a small set of subroutines are detected as being heavily used, and they are amenable to hardware implementation, a co-processor can be designed to provide the functionality of those subroutines. The exact set of co-processors, and what they explicitly provide can be left until much later in the design cycle. If a need is noticed for a co-processor to do four by four matrix multiplications because three dimensional graphics has become a common application, a specific co-processor can be designed and built for that task. An extra time cost in accessing the co-processor, if it exists, is introduced. It can actually be a large factor over the time taken for visible integration. The overhead in using a subroutine call and return with the required argument passing and value return can be much greater than the execution time of the co-process instruction itself. On a multiple processor machine, if the program is to be used on processors both with and without co-processors, some means of "swapping" which set of subroutines are used must be implemented. This may not be an easy task.

All approaches covered have undesirable features. What is desirable is to have the benefits of an invisible integration of co-processors, without the complexity of having the basic processor aware of the co-processors. Hiding of the co-processor by the use of subroutines is a preferred choice provided the overhead implicit in this solution can be removed, and that some means of integrating this with multiple processors is possible.

2.2.5 User Display

For time sharing systems the interface to the user has been what can best be described as primitive. While Teletypes have generally ceased to be used with computers, the interface to the user still tends to be based on the Teletype model.
Single user machines have provided a chance to spend more processing on the user than was possible with time sharing machines. This has lead to common bitmap display systems with a “point and grunt” style of interaction.

The “quality” of the display has become an important aspect of machines. It used to be important “what” a machine could do, but now it is equally as important “how” it shows you what it did. For a static display the higher the resolution the better. It is obvious that a graph on a display with 1024 by 1024 points is going to be more aesthetically pleasing than one on a display of 100 by 100 points. This argument implies that the larger the number of points available the better the display. For a dynamic display there is one other aspect of importance.

If changing one point takes 5 micro-seconds, the 100 by 100 display can be changed in one twentieth of a second. The 1024 by 1024 display will change in 5 seconds, which is important from a user's point of view. Using such a system would accent an alternate meaning of the term “methodical programming.”

Some means must be provided to allow rapid updates to high resolution displays. It should be possible to modify multiple points at once, thus reducing the total time needed to perform an operation. If, in the above situation, ten points can be modified at once, the 1024 by 1024 screen can be completely changed in one fifth of a second which is infinitely preferable to a five second delay.

If it is possible to change ten points at once, it would be perfect if any ten points could be chosen. The problems in both providing such a solution, and using such a solution are too great. It is obvious that if the ten points are clustered together that the solution is simpler to achieve, and simpler to use. For drawing horizontal lines it would be preferred if all ten points were adjacent horizontally. For drawing vertical lines it would be preferred if all ten points were adjacent vertically. For painting characters on the display it would be preferred if all ten points could contribute to the character. A good display should cater for all of these possibilities.

2.2.6 Time of Day Clock

The time of day clock is seldom considered an important part of a machine's hardware. Anyone who has dealt with the handling of time of day clocks, or has used a machine over a
period of time which covers a leap year when the problems were not correctly handled
knows that this is not true.

A time of day clock should handle leap years adequately. Not necessarily correctly,
only adequately. Carrying the situation to: every four, except every 100, except every 400,
... is excessive. No piece of hardware which will be built in the near future is going to have
to deal with the end of the 21st century. Dealing with every four years is totally adequate.

What is more important is the format which the clock uses to provide the time. The
time of day is used to mark many things within a system. At the bottom level it is used to
indicate file system times, at application levels it shows up in numerous places. If the format
of the time of day clock is not suitable for the uses for which it was intended, the format has
to be changed. A system standard format is easy to impose in a message passing system
where only one process deals with the hardware. If the system standard format does not
match the hardware format, a conversion has to be performed. Should it not meet the
requirements of the user of the data, a further conversion has to be done. It is preferable if
only one conversion were done. The hardware and system standard formats should be
identical. The file system would be best served by a concise format which would fit in one
convenient cell. A 32-bit word would be quite adequate. Kept as the number of seconds
since some arbitrary point, the file system would be served, but the conversion to human
readable form would require a considerable investment in software. The other inadequacy of
seconds since format is that all values are valid. The third inadequacy is that “don't care”
parts of the date are very hard to represent. When a person is asked what time it is, and
replies “9 o'clock”, it means that it is close enough to 9 o'clock that it does not matter what
the minutes and seconds are. Given a seconds since format this “about” is not possible.
The format chosen should support don't care parts. If the hardware provides the time of
day as a BCD string, as is common, it does allow don't care fields, but does not nicely fit
within a 32-bit word.

What is desirable is a 32-bit format which supports don't care fields, from hardware
that deals with simple leap years. By applying equivalent don't care fields two times can be
compared with word comparisons to yield a before, after or same time result. The time
format can be termed the close format since it encodes time to the precision of seconds.
This is adequate for most tasks however there are applications which need a higher
accuracy.
The accuracy of precise time is interesting. With a large amount of effort a nano-second incrementing clock could be built. It would not provide a time accurate to one billionth of a second. It takes time to detect that some event has happened, more time to respond, and even more time to obtain the “accurate” time. Accurate time is not generally needed to record a point in time, but to record a duration. For many real-time applications what is needed is an answer to the question, “How long has it been since ...?” rather than the exact point in time at which some event happened. A high frequency incrementing clock can be entirely separate from the time of day clock. In general it should be instantly available to all processes. Placing a process between the incrementing clock and the other processes which wish to obtain the value, would introduce an unacceptable variable delay in obtaining the value. If this clock can appear in a non-cached read-only segment of the address space of a process, it can obtain the value at any point in time it requires it.

As well as more accurate times, there is a need for a format which records a point in time from a larger range than is possible with the close format. The range in years which can be stored in a 32-bit word is limited if seconds are also recorded. Even if a seconds since format is used the range is less than 140 years. The historical format would consist of the close format, with the seconds field removed, and the extra bits made available for the year field. Given that there are 60 seconds in a minute, the range of years could be increased by at least a factor of 60.

The time of day clock provides the time in a close format. For applications requiring a larger range of years the historical format can be created from a close format by simple operations. For those applications which require high accuracy duration values a second clock will provide an instantly available high frequency incrementing value. The close format would consist of six fields in the 32-bit word, each an appropriate number of bits in length. The hardware time of day clock can be read as one 32-bit word, and can thus assure that all read times are valid.

Summary

This chapter has presented a direction for the rest of the thesis. As well as a direction, arguments presented here have lead to specifications of various aspects of the software and hardware. The operating system is going to be a message-passing based one. There will be provision for shared memory, but shared memory will not be a natural method of
communication. The message passing scheme is a variation on the Send-Receive-Reply form of message passing, where replies are optional, based on a known feature of the message. The messages will be of a small fixed size, with massive data movements being supported as a separate primitive. The area exposed for massive data movements is under the control of the process which is exposed. The memory management hardware will have the operational speed of a paging scheme, with the small overhead in space of a segmentation scheme. Multiple processors will be supported, and all need not be the same. The caching of data will be done on a logical rather than physical address. Co-processors will be integrated into the machine in a manner which will allow their easy and transparent addition and deletion.
Chapter 3
Interactive Aspects

The operating system cannot be truly designed without some interaction with the hardware
design. This statement can also be made with the roles reversed. One of the most difficult
tasks in presenting the design of a complete hardware-software system is that a part cannot
be taken in isolation and fully presented. The hardware is not designed to match the
software, nor is the software designed to match the hardware. Rather, the two are designed
step by step, in relation to each other.

The previous chapter has laid the groundwork for further design of both hardware and
software aspects of the system. This chapter presents some of the interactive aspects of the
two. In general, when two opposing views are held, the integration of those two views
requires concessions on both sides. The viability of that integration can be roughly qualified
by the sum of the "discomfort" felt by both parties. The less the discomfort the more viable
the integration. The perfect situation will arise when the views of both parties are not in
conflict. This is an infrequent situation. If it is realized that a stated view is really not the
true desired goal, it may be possible to reformulate the view in question so that there is less
conflict with other views. For example, a hardware view that paging is more desirable than
segmentation is one presentation of the hidden fact that address translation can be more
swiftly performed when the operations required do not require arithmetic operations. The
software view that segmentation is more desirable than paging is one presentation of the
hidden fact that sharing of parts of programs is conceptually simpler with a logical division
of the address space. Integrating the hidden facts may be simpler than integrating the stated
views.

Section 3.1 Memory Management

The basic tasks of memory management hardware are address translation, and invalid
addressing detection. For each address presented to the memory management hardware it
must change a logical address to the corresponding physical address if the address is valid,
or indicate that in the current context the address is not appropriate. The hardware designer
would prefer that these tasks be handled in the simplest and fastest way possible, so that
any perceived shortcomings of the complete system can be laid at the feet of the software
designer. The software designer would prefer that these tasks be handled in such a way that
any perceived short comings of the complete system can be laid at the feet of the hardware
designer. When both hardware and software are designed by the same person there is only one pair of feet.

The operating system should support multiple communicating processes. A process should be "inexpensive". That is to say, if the hardware for memory management was based on a paging scheme, with pages containing 1,024 addressable units, within a 32-bit address space, and required that the paging tables be completely defined for each process, there would be a requirement for over four million page table entries for each process. This would not be desirable if the process in question was a simple timer process which used only a few hundred addressable units of storage. The existence of such a process would definitely not be considered "inexpensive" in this situation. If the page size were 65,536 addressable units long the process could also not be considered "inexpensive" because of the internal fragmentation of memory. From the operating system view point, a process should only necessitate the use of a small amount of memory over that directly occupied by the program the process is using. Because the basic paradigm of the operating system is many small processes, communicating to accomplish a required task, this implies either a paging system with a small page size, or segmentation, as the best solution. A small page size is in conflict with the further requirement that extremely large programs also be supported, since this would necessitate an extremely large number of page table entries. A reasonable position to take to start this discussion is that the operating system would seem to imply that segmentation is a desirable mode of operation for the memory management hardware.

Assuming that segmentation is the chosen mode of operation, it is necessary to consider how the operating system would view the memory model. A process uses some program which has been created for its specific task. At least one segment must exist for this specific program. It also will probably make use of functions which can be taken from a common library of functions which are not task specific. At least one other segment for these library functions is useful. It manipulates data which are specific to the process in question, though these data may initially be set to some common values specific to the program it is using. One more segment to support the process specific data items is needed. Given that the process communicates with other processes by means of sent messages, and that the copying of these messages should be avoided, this introduces a fourth segment. For such processes as device drivers, some means of accessing the communication area of the device is required, which introduces yet another segment. If shared data is supported, there is a need for a further segment for the shared data, to keep it separate from unshared data.
One of the real difficulties with a segmented memory scheme is that the segments are not of equal lengths. When a new segment is needed, or one changes in size, an appropriate area of memory must be found for it. Not only is searching for an area required, but compaction will also be implied. The compaction requirement has implications for the potential use of direct I/O to the address space of a process. The major impact on the hardware from a segmentation scheme is the apparent necessity of arithmetic operations to convert logical to physical addresses, and to check segment limits. Of the various classic memory allocation methods, the buddy system [Knuth 73] is worth considering.

If every segment is some power of two in length, both hardware and software aspects of segmentation are simplified. The need for arithmetic operations in the memory mapping hardware are removed. For example, consider a segment which is 1024 addressable units in length. The segment is located at a physical location which is a multiple of 1024. The valid range of offsets within the segment is 0 to 1023. The computation of the physical address is a bitwise "or" of the segment base and the offset since there can be no overlap in non-zero bits. If the valid range of addresses is viewed as bit patterns it can easily be seen that all non-zero bits of the offset must lie within the least significant bits of the address. The complement of the valid set of bits provides a mask which can be used to identify an invalid address. Each segment has two descriptive values, a base address, and an offset mask. The computation of the physical address is:

\[ PA = \text{BASE}\{\text{Segment}\} \mid \text{Offset} \]

The detection of an invalid address is:

\[ \text{INVAL} = \text{Offset} \& \sim\text{OFFSETMASK}\{\text{Segment}\} \]

Both of these are trivial operations in hardware.

Buddy system segment allocation provides much of what is desirable, from a hardware point of view, that a paging system provides. Expensive arithmetic operations which can involve carry propagation are avoided, and are replaced with simple bitwise logical operations. The computation of the physical address and the validity of the address can be done in parallel and so the additional offset validity check need not slow the memory management operations. With a paged system, it is not the offset which can invalidate an address, but the page number. If it is possible to allow all segments to be valid, some with no valid offsets, there is no need to check the validity of the segment number since invalid segments can be caught by invalid offsets. The previous statement is not quite true, since an offset of zero into a segment would pass the offset validity test. The simplest way to
overcome this, is to assume that the least significant bit of the offset from the logical address is a one when computing the invalidity of the offset. Only with an offset of zero, and a segment length of zero, will there be an effect in the result, and the effect is exactly that which is desired.

Given either a paged or segmented system, one common point is that, since the logical address is split into an offset and a segment or page number, some decision has to be reached as to where in the logical address the division occurs. One extreme can support a large number of segments or pages, while the other extreme supports a large offset. With paging, this is not of great interest to the programmer because the division is supposed to be hidden from the programmer. Segmentation presents a problem, for the very reason that the split is not hidden. Given a large number of segments, the maximum size of any segment is reduced. Taken too far, it can prevent the programmer from using large memory structures, as one array must fit within one segment, as well as implying a potentially large segment table. Taking the other extreme, large structures can be supported, but a buddy scheme for memory allocation implies that should a segment need to grow it may double in extent. If the segment were only 128 long, doubling to 256 is not too bothersome, but doubling from 131,072 to 262,144 is a noticeable jump in size. Multiplicative growth is one disadvantage of using a buddy system. More will be said about this later.

Apart from changing logical addresses to physical addresses, and checking that the address is valid, the memory management hardware should also check that the access requested is valid. It does no good to claim that the program segment of a process can be shared since it is not modified unless any attempted modifications to it are detected and prevented. Segments can be classified as either modifiable or not. A generic permissions model tends to include at least three permissions. A process can be given or refused the abilities to READ, WRITE or EXECUTE on a segment by segment basis. It has already been shown that a controlled WRITE access is both desirable and necessary. The other two require further investigation.

What reasons can exist for requiring control over READ access? In general there are two. A process attempting to read a location for which there is no need to read can possibly be using an incorrect program. Detecting such an attempted read operation would indicate that the program is in error and that it should be repaired. While an admirable position, it does not guarantee that the program is correct because such an attempt is never made. Nor does it assure that the invalid read will occur close in either time or space to where the error
occurred. Detecting invalid programs by the use of READ protecting segments does not seem to be extremely useful. One area which could make use of this is the protection of proprietary software. An unscrupulous programmer could "read" the instructions of a piece of software and "disassemble" it back to a source form to gain access to proprietary information. This is easily done in a system which links the software into the address space of user programs. A message passing system provides another means of allowing the use of proprietary software. If the proprietary software is provided as a supporting process, the program is not in the address space of any process which makes use of it, and is by definition not readable. The use of a READ attribute seems to be unnecessary.

A process has access to executable code and to data. All the executable code obviously must have EXECUTE permissions. The data however can be protected against execution as instructions. The chances of accidently attempting to execute data by any program written in a language which does not support programmer initiated indirect function calls are very close to zero. Introducing extra complexity into both the hardware and software to prevent this appears to be unnecessary. Doing so also introduces extra complexity into programs which want to generate executable instructions as they operate. For example, in a program which performs massive amounts of pattern matching it can be very advantageous to generate a small piece of executable code to match a specific pattern rather than make use of a general pattern matching algorithm. Such a program would need a segment which could be written to, and also executed from. It seems easier to allow EXECUTE for all segments.

The only control that appears to be necessary is that over WRITE access. A separate indication of the access desired could be given to the memory management hardware and each segment could carry an indication of whether or not the specific segment was modifiable. A simpler scheme would be to make the ability to write to the segment implicit in the segment number itself.

If exactly half of the segments can be written to and the other half cannot, one of the bits of the segment number in the logical address can carry the write permission flag in it. If the most significant bit of the segment number contains the write prohibit bit, a simple AND operation of the requested access with the most significant bit of the segment number could indicate the validity of the access. Segments which should be protected from write access must be in the upper half of the address space. These segments would contain the
executable functions of the programs the processes use. Apart from being slightly unconventional, there is no reason this should not be so.

Earlier, more than four, but less than eight segments, were tentatively identified. Eight segments is a reasonable number to use for initial consideration. The 32-bit logical address consists of three parts. The least significant 29 bits contain an offset into the segment. This supports up to 536,870,912 addressable units in any one segment. The most significant bit indicates that modification is not allowed. The three most significant bits indicate which of the eight segments is specified. The uses for each segment can now be considered.

A large number of the direct references to memory locations by a program are references to data which were defined at compilation time. It would thus seem reasonable to assign the compilation time data to segment number zero. If the instruction set provides for some means of specifying an address in a shorter form than the full 32 bits, this would appear to reduce the size of the code of a program, since many references to compilation time data could make use of the shorter format.

A large number of function calls will be to functions which are provided in the program specific to the process so it also seems reasonable to assign the first non-modifiable segment to the program specific code. If there is a short form of addressing to support function calls within the first part of the first segment, many calls could make use of the short form, further reducing the size of the program.

If a non-copy mode of message passing is used, another modifiable segment can be assigned as the message segment. Three of the eight segments have been assigned as can be seen in figure 3.1.

When library code is considered we have three free segments which are not modifiable. It seems reasonable to only use one for the library code. Exactly how the library segment is to be used will be covered later. It is sufficient for now to assign it to one of the non-modifiable segments.

While there are a large number of programs which do not make use of dynamic memory allocation there are also a large number which do. Two possibilities are apparent. The dynamic memory can be "tacked on" to the compilation time data segment, or it can be allocated a segment of its own. There is no real way to make use of short addressing modes when dealing with dynamically allocated memory since the location of the allocated
memory, and hence the size of the address, is only known at execution time. No instruction size saving can be made by attaching the dynamic memory to the static sized compilation time segment. Attaching it to the compilation time segment can have one serious effect. Since memory is allocated using a buddy scheme, a small amount of dynamically allocated memory may force a doubling of the size of the data segment. If doubling can be avoided by using another segment, it should be. It is worthy of note that, if this is the chosen method for dealing with dynamic memory allocation, both the first modifiable and non-modifiable segment sizes are completely under the control of the programmer.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Write</th>
<th>Use For Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>Compilation Time Data Segment</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td>The Message Segment</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td>Program Specific Code Segment</td>
</tr>
<tr>
<td>5</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1 Initial Segment Assignment

Given a situation where the classic space/time trade off is possible, the programmer has the information available to make reasonable choices. If the size of the instruction segment is larger than the size of the instructions, the programmer is at liberty to use more instructions to gain a speed credit without incurring a space debit. Should the size of the instructions be slightly more than half the instruction segment, a minor reduction in the space used can gain a major space credit at the cost of a minor speed debit. Inspection of the usage of the compilation time data segment, may permit the programmer to make use of some static buffers which would usually be allocated and thus save in total space usage. If all the dynamic allocation could be reduced to static allocation, a simplification of the program, and reduction in size, would also be possible. Another possibility is to “table drive” certain parts of the code, trading more data space for less code space and less time used.

One of the disadvantages of the buddy scheme for memory allocation is the doubling of memory usage with a minor increase in need. It may, as well, be viewed as one of the
benefits of the buddy system. Ignoring for the moment the “get it out the door and make a buck” requirements of commercial programming, a programmer has a moral duty to the users of the software produced to do as good a job as the programmer is capable of. The programmer must take responsibility for his/her work. This is true in all occupations. The production of software is slightly different since programmers produce “black boxes” for others to use. A poorly built engine in a car is visible for inspection to the purchaser, and it wears with time and will soon break. Software is hidden from the purchaser and does not wear out. It will continue being as bad as it was when it was produced. By causing a large effective difference from a small true difference the buddy scheme can provide an inducement for programmers to understand some amount of the programs they produce.

Three of the four modifiable segments have now been identified, but only two of the four non-modifiable segments. There is a very reasonable use for another non-modifiable segment. Many of the requests made by processes of the operating system fall into an inquiry category. Questions such as “Does process X exist?” and “Which process is the file system?” are commonly asked within a message passing system. Some of these questions must be asked every time the information is needed, while others are, in general only asked once, and the information “tucked away” in some program variable for later use. For those questions which must be repeatedly asked, it would be advantageous if an inexpensive means were available for any process to obtain answers to such questions without asking. Providing such a “billboard” in one non-modifiable segment to every process would allow swift access to the answers to questions which must be repeatedly asked. This means of swiftly obtaining the answer is also of benefit for those questions which have answers which never change, such as identifying the file system process. By tucking away the answer to the question when it was first asked, a program can benefit by not having to ask a slow question to get an answer it should already “know”. The disadvantage with this internal billboard is that later modifications of the program may result in the “fast” answer being used before the question is asked. Such subtle bugs in programs become hard to find, since the modifier probably does not have all the intimate knowledge of why the program does, what it does, the way it does it. Providing a fast answer to questions that need only be asked once allows the question to be asked every time the answer is needed, and reduces the urge of the programmer concerned with “efficiency” to lay a trap for later maintainers. The third non-modifiable segment provides a system billboard which allows access to inquiries that are commonly made.

Six of the eight segments have been assigned a specific use as seen in figure 3.2.
Neither device drivers nor processes which share memory have been dealt with at this point. There are two segments which are still free, and can be used to satisfy the requirements of these two special cases.

A device driver needs access to the device communication area. Access is for both reading and writing. A modifiable segment can be used by device driver processes.

For processes which share memory there are two basic classes into which they fit. A process which has access to a shared segment of memory either must modify it, or it must not. If the access required does not involve modification, the last read only segment can be used to give the process access to the area in question. If modification is required there appears to be no free segment which can be used. Either one of the assigned modifiable segments must be revoked and reassigned, the number of segments increased, or the division of segments into four modifiable and four non-modifiable segments must be changed.

To change the division of the segments from four modifiable to five modifiable is possible but not desirable. Doing so implies that the identification of which segments are valid targets for write access would become slightly more complex. This is the least desirable solution since it increases complexity.

Increasing the number of segments is possible but also not desirable. With a segmented memory scheme either a small number or a large number of segments is best. With a very large number of segments, programs can be written to use the segments in an
effective manner. If, for example, there were enough segments so that every function could be in its own segment, the possibilities of dynamic loading and updating on a function level become available. This is not a new idea, having been seen in Multics. With a minimal number of segments there is just enough to support the programs and the programmers can think of the address space as, essentially, contiguous. With an intermediate number of segments, the programmer is presented with smaller segments than possibly desirable, but too few to truly make use of these segments as separate units. Increasing the number of segments only slightly brings no true benefits.

It is possible to reassign one of the modifiable segments for use as a shared segment. The last modifiable segment is only used by device driver processes. As shared memory is supported only as an extra feature for those tasks which have very heavy information flow, or need access to large common bodies of information, it need not be used by device drivers. The fourth modifiable segment can be assigned as the modifiable shared segment for processes which share memory. Shared memory if fraught with subtle traps for the unwary and making it special may well reinforce care. The final assignment of the eight segments can be seen in figure 3.3.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Write</th>
<th>Use For Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>Compilation Time Data Segment</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td>The Message Segment</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>Dynamic Memory Segment</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>Device Driver/Writable Shared Segment</td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td>Program Specific Code Segment</td>
</tr>
<tr>
<td>5</td>
<td>✗</td>
<td>Library Code Segment</td>
</tr>
<tr>
<td>6</td>
<td>✗</td>
<td>System Billboard Segment</td>
</tr>
<tr>
<td>7</td>
<td>✗</td>
<td>Readable Shared Segment</td>
</tr>
</tbody>
</table>

Figure 3.3 Final Segment Assignment

Of the eight segments that each process can use, not all are specific to that process. Every process will have a compilation time data segment and a message segment which are unique to that process. Because multiple processes can use the same program code segment a program specific code segment need not necessarily be unique. Only those processes which use dynamic memory need a dynamic memory segment. The library code segment is shared by all processes, as is the system billboard segment. Only device drivers need a
segment which will allow access to a device communications area. The few programs which use shared memory will have access to one or two of the shared memory segments.

The library code segment and device communication segments are not allocated segments. The device communication segments simply are access mechanisms so that the physical device communications areas can be mapped into the address space of the device drivers. The library code segment is stored in memory which is possibly distinct from the normal allocatable memory. As such it too is not allocated. In a mature system the library code segment may well be stored in ROM. In a developing system this ROM area of physical memory may be provided as RAM which is loaded at initialization time with the appropriate sets of functions.

The system billboard segment is allocated once. All processes can be assigned access to that allocated area. The dynamic memory segment is allocated on the first dynamic request of a process, and may change in size with time. The shared memory areas are allocated on the request of the processes which use shared memory. Since these are assumed to be few, the creation of a new process normally involves the allocation of only two segments of memory, the compilation time data segment and the message segment. It is assumed that the loading of the program specific code segment has occurred earlier.

Justification for the use of a buddy scheme for memory allocation was based on simplicity of the hardware support needed. The implications of such a scheme must be considered in more depth.

Since the buddy scheme works by finding the smallest division of memory which is a power of two in length, which is at least the size of the area required, no segment can be bigger than half the memory available. Given the normal size of physical memory in current machines this appears to be an inconsequential restriction. The true problem with a buddy scheme, or any segmentation scheme, is that compaction may be required to satisfy a request for a new segment, or the increase in size of a segment.

Compaction implies finding the appropriate pieces of occupied memory that should be moved to make a hole of the required size. Some sort of reasoned method should be used to identify these pieces. For example, if 1024 units are needed and there is a 512 unit hole, one choice is to move those pieces in the 512 unit buddy. The method would assure that no more than half of the required hole size would have to be moved. While sounding
It quickly became obvious what the best algorithm was. If the memory could be “visualized” as a vertical set of horizontal rows of the appropriate width, the row to empty was obviously the row which was least full. A program cannot visualize, but it can put a number into an entry of a vector which represents the “amount of black” in that row. The index of the minimum entry in the vector indicates which section of memory to empty.

The algorithm is simple to implement. Pass over descriptors for all allocated segments, setting the values in the entries of the vector. With the minimum entry identified, all segments which are represented by that entry are moved so that they would be
represented in some other entry. No segment will contribute to more than one entry unless
is completely fills all entries it contributes to.

If all segments which should be moved have alternate locations available to them,
moving them produces the appropriate hole. Should they not be moveable because they are
larger than any available hole, the algorithm has to be repeated for those segments, using an
appropriate “display width”. Before finding the minimum entry in this new vector all entries
representing the area currently being emptied are set to “full” so they will not be candidates.
This avoids a potential infinite loop. There is one interesting twist here. It is involved with
the order of movement, and completion of the algorithm.

Consider trying for a 8192 unit hole. Two entries have 1024 and 1080 for values.
One segment contributes 512 to the 1024 entry, and one segment contributes 128 to the
1080 entry. To move the 512 segment the 128 segment has to first be moved. When the 128
segment is moved, the 1024 entry is still 1024 but the 1080 entry is now 952. To move the
512 unit segment, and the others to complete emptying the 1024 entry, would involve more
work than restarting and emptying the new 952 unit entry.

The more detailed version of the algorithm would suggest attempting to move the
largest segments first. If a segment cannot be moved, the algorithm is recursively used to
identify and move one segment. For example, to move the 512 segment the 128 segment
has to be moved, which might require a 64 unit segment to be moved, which might require
a four unit segment to move. That four unit segment is the one that is moved. After the four
unit segment is moved the algorithm restarts with identifying the 8192 unit entry that should
be emptied. Because one segment has moved the whole solution may be different as the
above example showed. Some times the sub-sub-problem is the only one affected. Perhaps
after moving the four unit segment the 64 unit segment becomes not the best to move to
make room for the 128 unit segment.

To test that the algorithm given below does a reasonable job of compaction a
simulation of 10,000,000 allocations and deletions of “segments” was carried out. Every
time compaction was necessary an exhaustive search of all reasonable movements was
carried out to identify the exact minimum amount of movement necessary. The algorithm
was then used to perform the movements it identified as required. At the end of the
simulation, which took over forty hours of processor time, this algorithm had moved
exactly the minimum amount. The simulation does not amount to a proof of optimality of
the algorithm, but is a reasonable indication that it is acceptable as far as movement is
concerned. The algorithm is not optimal. Pathological cases can be created for which it does not choose the best solution. Such situations just did not arise. In the vast majority of cases, the first entry to be identified as the one to empty, turned out to be the one to empty. Only when the difference between the minimum entry and the next smallest was extremely close did shuffling at a smaller unit size change the problem.

The efficiency of the algorithm itself is reasonably high. The steps are quite simple.

1/ Make a pass over the vector, setting each entry to zero.
2/ Make a pass over the allocated segment descriptors, adjusting the entries in the vector.
3/ Make a pass over the vector to identify the minimum entry.
4/ Make a pass over the allocated segment descriptors to find the segments involved in the minimum entry, and sort the list of these segments by size.
5/ For each entry in this sorted list, attempt to move the segment.
   5a/ If the segment could not be moved, find and move the smallest segment which would eventually lead to this segment being moveable, restart the algorithm from step 1.
   5b/ If the segment can be moved, move it.
6/ Terminate with a free segment as big as required.

Most of these steps are O(n). The sort is at worst O(n^2) but the list of segments to be moved is much shorter than the list of all segments. When re-application of the algorithm is necessary to identify a smaller unit to empty the algorithm becomes more expensive. This appears to be an extremely small percent of the time, fortunately. The worst case should be considered. Every segment which has to be moved can require the movement of another segment half as big, all the way down to the movement of a one unit segment. The algorithm could take up to log_2(size) times the best case.

There can potentially be cases where this algorithm will fail to create the appropriate empty section. In order to make sure that the algorithm does not get into an infinite loop, shuffling segments back and forth, the sub-movements are restricted in the areas to which they can move segments. It is possible that all potential areas can be “locked” before any segment can be found that can be moved. Should this situation arise, a solution is to chose any segment which can be moved closer to the start of memory, move it, then retry the algorithm. In the worst case every segment will have to be moved, but that will create the needed empty space. Before this brute-force approach, it is worth trying to move one of the segments identified as needing to be moved, but impossible to move under the constraints given. During the simulation mentioned this occurred only eight times, and the first segment which could be moved happened to be one which resolved the problem.
Section 3.2 Cache Memory

There are three major reasons for introducing cache memory into computer system. One is that it provides faster apparent access to slow memory. A second is that it can reduce the contention for the bus in a machine where more than one piece of hardware can use the bus. A third reason is that cached memory can serve as an alternate to a larger number of registers.

If the only reason for introducing cache memory to a machine is to provide faster apparent access to slow memory, this might no longer be justifiable. The speed of normal memory has been increasing and the differential between common memory and the affordable faster memory technologies has been decreasing. In certain situations cache memory may be of benefit but in most cases the extra cost of cache memory may be more profitably spent in obtaining slightly faster main memory storage.

When multiple bus masters exist a much more compelling reason exists for cache memory [Briggs 83]. Because there can be more than one "processor" attempting to use the bus at any given time, contention for the bus can arise. While the bus is in use by one processor, the other processors which are attempting to use the bus are forced to wait. It is this very contention which puts a practical limit to the number of processors which can be used in a multi-processor machine which uses a common memory area. As the number of processors increases the potential increase in useful work which can be done per unit time increases. At some point adding another processor, while allowing yet another task to execute in parallel with others, causes enough contention for the bus that the effective work that each processor can do decreases. Carried to an extreme, the useful work done on a multiple processor machine can be reduced down to less than what one single processor could have provided.

The third reason for introducing cache memory, the replacement of a larger number of registers, is also important. A cunning compiler, given a large number of registers to work with, can arrange that the most frequently used variables will probably be allocated to registers. This can increase the speed of a program noticeably. What a cunning compiler can most probably not do is to detect that two memory locations with computed addresses, for example two references to locations within an array, will be references to the same location during execution. The compiler can arrange that the value of one entry in the array, say
Array[i], can be held in a register so that a further reference to that location can be redirected to use the contents of the register, but when any entry in the array is changed the register contents may no longer be valid. The compiler can only assign registers based on compilation time addresses. Cache memory works on execution time addresses. It is immaterial how the address is computed, whether by direct name known at compilation time or as the result of a complex expression, since an address is an address. The provision of cache memory can reduce the time needed to access the same location repeatedly, when the compiler cannot detect that the addresses are the same. Another problem with compiler allocated registers is that the compiler has to decide with static information which locations are most frequently used. It can "know" how many references to a named location occur in the code, but it can only estimate the number of times each reference will be used. Compiler assigned registers may very well reduce the size of the code generated, but most likely will not be as effective at reducing the time taken to execute that code.

If the reason for introducing cache memory is only to accommodate slow main memory, the cache memory can be placed on either the processor or memory sides of the bus. Simplicity is a major reason for choosing to place the cache memory with the memory it is supporting. Another reason may well be for configurability.

By placing the cache with the memory a machine can be given what appears to be faster memory by replacing the memory unit with one containing more cache. Were there more than one bus master, which is commonly the case, all would benefit from the increase in access speed. When the cache is with the memory, it is also easy to have the cache treat stores as having happened long before the value has been passed to the memory. This can decrease the apparent memory write time significantly. It is not as easy if the cache is with the processing units.

Simplicity is by far the most compelling reason for placing the cache in the memory unit. Consider a machine with two processing elements, each with its own cache. The cache either works in a write through mode where every memory write is immediately passed through, or it holds the write as a pending write until a more convenient time. A write through mode of operation will obviously cause more bus contention if the location were updated frequently. Consider for example the variable which is to contain the maximum value of an array, which just happened to be sorted. Each test of an array entry would result in the variable being changed. If the change to that variable could have been left as a pending write, only one write to memory would have been needed. If a pending write
cache is used there is an immediate problem with more than one cache. First the write through mode will be covered since it is a simplified version of a pending write mode.

A write through cache, when there are multiple bus masters, must monitor all stores which are on the bus. If it detects a store to a location which it is currently holding, it must either update the value it has, or “forget” that it is storing that location. Failing to do either would result in a potential effect in the computation, induced by the cache itself. Which of the two actions would be the best is impossible to know in general but one or the other must be taken. A write through cache must work with physical addresses and monitor all changes which appear on the bus. Being placed with the processing unit, it can reduce bus contention by providing a short cut for memory read operations only.

A pending write cache has extra complications. In order to maintain its contents it must monitor the bus as does a write through cache. It is also responsible, however, for the contents of any locations that other caches may have. A write through cache monitors the bus in a passive mode, noting changes to memory locations. A pending write cache must monitor the bus actively. If it detects a memory read for a location which it is currently holding as a pending write, it must intervene and provide the value that the memory would otherwise have had. This can complicate the architecture of the bus, since there must be some provision for this active intervention. There has to be some sort of, “Let me get that for you,” indication either through extra control lines or by means of logically disconnecting the memory from the bus for the appropriate memory reads. The situation is also further complicated when a memory write operation is attempted. If no other cache is holding the value of the location being written to, the write can be treated as a pending write with no further problems. If some other cache does have the location, the pending write must be treated as a write through, otherwise the other cache will be left containing an incorrect value. If one cache is to know exactly which locations any other cache has that it also has, it must be informed not only when the other cache obtains the value of that location, but when it no longer has that location stored. Propagating these cache flush locations would use a reasonable amount of the bus bandwidth, and storing the information within each cache would take a reasonably large amount of space. The only viable solution would be to remember if any other cache could potentially have the same location stored. When the location was loaded there would have to be some indication that the location was loaded from memory, or that some other cache provided it. When the location was stored without first being loaded, the write would have to be treated as a write through and some indication would have to be available that some other cache took a copy of the stored value.
The benefits of not having to pass all writes through to the bus are rapidly being eroded by excessive complexity and communication amongst the various cache memories.

It would seem that the only viable alternatives are to place the cache with the memory, or to use the cache in a write through mode, but in either case the cache would have to function with physical addresses.

All of the previous discussion has been based on one major assumption. It has been assumed that two caches can potentially hold the same location, and that the location can be modified. Consider a message passing system with processes having modifiable address spaces which are disjoint. Whether the address spaces of the processes share a non-modifiable segment or not is immaterial. In such a situation no two processes will ever reference the same physical location for modification. No two caches will ever hold the value of the same modifiable location. No cache need pay any attention to modifications made by any other cache. No cache need inform any other cache about modifications it either makes, or has pending. Whether it operates in a write through or pending write mode is important only locally. Further, whether it works with logical or physical addresses is a local consideration as well. If such a system is the case, complications introduced by multiple bus masters can be ignored when cache memory is being designed.

Of the eight segments provided by the memory management unit, such a situation occurs for only five of them. The three segments which do not satisfy these requirements are the readable shared memory segment, the system billboard segment and the segment which is either a modifiable shared memory segment or a device communication segment. Keeping of the values in the device communication segment is ridiculous, but keeping of these segments for other uses requires discussion.

The system billboard is provided so that processes have a convenient and efficient way of obtaining reasonably accurate information. If the segment is not cached, the decrease in speed is probably not going to be too great. It would certainly not be as great as it would if the information had to be obtained in some other manner. Access to this segment is also probably going to be so infrequent that any cached locations are probably going to be lost before they are accessed again. Another consideration comes from a much earlier discussion about the time of day clock and the need for a duration clock. This system billboard segment is just the right segment to include the incrementing clock.
The shared memory segments are provided for the express purpose of allowing changes in shared locations. It should be relatively safe to assume that any degradation in operation of processes which use shared memory can be recovered by the benefits of using a pending write cache mode, if not within these very same processes, then overall.

It has been mentioned that a cache in such a convenient situation could work with logical addresses equally as well as with physical addresses. In certain situations it could work more effectively. Given that at some point in time the memory allocation operations are going to require compaction in order to obtain a required free segment, the logical to physical mapping of some segments for some processes are going to change. Should one of the processes in question be active, it must be paused while this compaction is happening. It is quite possible that the segment being moved is the program specific code segment. As such, duplication of the segment while moving it does not invalidate the old copy. The old copy of the segment is only invalid when it is freed after duplication. In such a situation, with a cache using physical addresses, all cached locations in that segment would then become invalid, if for no other reason than that those physical locations will never again be referenced. The cache will have to slowly fill with references to the new physical locations. A logical address cache is unaware of the displacement of the logical address space in the physical address space. All cached locations in that segment are still valid. There is no need to reload anything. For modifiable segments the situation is not as simple since the process will have to be paused for the changing of the memory management registers and for the full duration of the copy as well. Despite this slightly longer period of time, the cache still need not be informed of the change in any way.

Of the five segments which can be cached with logical addresses one is most surely going to be changed whenever the process blocks. This is the message segment. In general the only way a process would block in a message passing system is by attempting to communicate with some other process, which is going to involve an exchange of message segments. When active the number of references to the message segment are going to be few. When preparing to send a message or give a response it is going to set some number of the fields in the message. It will likely set each field once. When it has received a message or a response it is probably going to access the fields a few times each, and more than likely that number is going to be one. Any holding of this segment is probably useless. This can allow the number of segments which are cached to be reduced to four which is a very convenient number. The most significant bit in the segment number indicates to the
memory management hardware whether the segment can be modified or not. The next most significant bit could very well indicate whether the segment should be cached or not. This requires reordering the segments slightly, as seen in figure 3.5. The cache memory hardware need only manage locations which come from those areas where cache memory would be of benefit, and for those areas where it would not, the cache memory could immediately pass the locations through.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Write</th>
<th>Cache</th>
<th>Use For Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>Compilation Time Data Segment</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>Dynamic Memory Segment</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✗</td>
<td>The Message Segment</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✗</td>
<td>Device Driver/Writable Shared Segment</td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td>✓</td>
<td>Program Specific Code Segment</td>
</tr>
<tr>
<td>5</td>
<td>✗</td>
<td>✓</td>
<td>Library Code Segment</td>
</tr>
<tr>
<td>6</td>
<td>✗</td>
<td>✗</td>
<td>System Billboard Segment</td>
</tr>
<tr>
<td>7</td>
<td>✗</td>
<td>✗</td>
<td>Readable Shared Segment</td>
</tr>
</tbody>
</table>

Figure 3.5 Cached Segment Assignment

**Section 3.3 Multiple Processors**

Having more than one processor in the machine has a few ramifications. Some of these have been covered earlier as is inevitable when discussing interaction. This section will concentrate mostly on the interactions between multiple processors and the kernel of the operating system.

Given that the machine can have some number of processors, where that number can differ over time, there has to be some way for the kernel to discover how many processors exist. Further, not only the number of processors but which processors exist is just as important. Considering that all processors need not have the same set of co-processors, nor execute the same instruction set, there also has to be a way to discover exactly what each processor implies. Also involved are the complications introduced by failure of a processor. If the information that the kernel reads can only be modified when the machine is running, and the failure of a specific processor causes the machine to not run, there is a “Catch-22” situation.
Making it possible to modify the configuration information, without the normal operation of the machine, can be done in many ways. This information can be “stored” in the positions of various physical switches. A maintenance mode of operation can be designed and built in. A special diagnostic processor can be included. The memory storing the configuration information can be a piece of non-volatile memory that can be modified in some manner external to the machine. All of these are worth considering. Each has costs in money, complexity and inconvenience. Each has its own benefits. The wrong question has been asked however.

The whole discussion has been based on the assumption that the kernel has to “find out” what processors exist. This assumption uses the active mode. Rephrased in the passive mode, which is the natural mode for servers of any kind in a message passing system, the question becomes, “How is the kernel told which processors exist?” and this is much simpler.

Assume that there exists configuration information for each processor in some known format and location. It is obvious that when a processor is being used by a process, and that process becomes inactive for some reason, the kernel must obtain knowledge of this fact. Here it is also obvious that the kernel needs to be told about the situation. Some mechanism must exist for the kernel to be told about a processor becoming available. When the machine starts no processor is available. As each processor successfully completes its initial checks it uses the mechanism to announce that it has become available. The only difference to the kernel is that when a processor becomes available for the first time, there is no process that it need record as becoming inactive. Turning the machine off and removing a processor simply means that processor cannot tell the kernel that it is available, and the kernel will never need to know that it ever was. Adding a processor requires setting the configuration values for that processor as required, turning the machine off, adding the hardware, and turning the machine back on.

If the hardware is built to withstand the transients induced by making and breaking connections while powered, the situation is even more pleasant. To add a new processor the configuration information needs to be set, then the processor added. To the kernel it is just a processor becoming available. To remove a processor, set its configuration information to some specific value which indicates that it is not available, force it to become idle, and then pull it out. Since the configuration information would never match something that the kernel would be looking for, no process would ever get assigned to it.
Configuration information has been mentioned in the most vague terms so far. It is now time to be a little more specific. This information must indicate the type of the processor, and which styles of co-processor it has. If the phrase "ignorance is bliss" is true, the kernel is indeed in a state of bliss. A process is assigned to a processor. The process uses a program. If there are three pieces of information attached to each program that the kernel can use, it need know very little. These three pieces of information are a MUST HAVE mask, a MUST HAVE value and a WOULD LIKE set. Consider that the configuration information consists of one 32-bit word. The instructions in a program force a specific instruction set. The MUST HAVE mask must cover the bits which include the processor type, and the MUST HAVE value must indicate the processor type. The WOULD LIKE set would cover the bits assigned to the appropriate co-processors that the program can make good use of. This need not be exactly the case. A given program may include some set of co-processors in its MUST HAVE mask and value, if that is desired. This may be done, for example, so that software needs a physical "key" installed to be run. That key is a specific co-processor.

To determine which processor to use for a given process the kernel applies the following algorithm to each configuration word.

\[
\text{if} \ (\text{configuration} \& \text{MUST\_MASK}) \neq \text{MUST\_VALUE} \\
\text{fit} = 0; \\
\text{else} \\
\text{fit} = \text{Sum\_bits} (\text{configuration} \& \text{WOULD\_LIKE}) + 1;
\]

This gives a value from zero to 33 to each processor. Processors with a fit value of zero can be ignored. The other processors can then be ranked by fit value. The algorithm for finding the processor to which the process will be assigned can be as complex as needed. Obviously if there is a free processor with the largest fit value, that is the processor to be assigned. At no point does the kernel "know" what the configuration word means, only what to do with it. It is desirable if the processor type values for the various processors all occupied the same locations in the configuration words so that the kernel's operations would be meaningful, but how big that area is, and where it is located, is not the kernel's concern. It is also worth noting that for a processor of type 6 the most significant bit might indicate a floating point co-processor, while for a processor of type 15 the bit might indicate a four by four matrix multiplication co-processor. For a type 9 processor the same bit could indicate that the processor is a new revision with some feature fixed. Should some
programs only work with the new revision of that type of processor, those programs could include this bit in their MUST HAVE mask and value.

If a process is assigned to a "less than best" processor, the only difference from the best choice will be a degradation in performance. Depending on what was not available, and how much it was of benefit, this degradation can be anything from not noticeable to annoying.

It has been decided that a processor "tells" the kernel that it is available. Some means must be found to do so. The simplest means seems to be having a unique location assigned to each processor. Into this location the processor writes its availability indicator. The kernel can read the location to discover that the processor is available. This is easy to do, but requires the kernel to poll common locations. Polling common locations would consume much of the bandwidth of the bus.

A far better solution would be to have one location, accessible over the bus, which would place an indication into some area which was specific to the kernel processor. While this location could be a set of locations, one for each processor, a single location would be more preferable when viewed from the kernel's position. If the bus accessible location allowed writing to a FIFO, the processor could place its identification there. The kernel need only read from the FIFO and would get the identification of each free processor, one at a time. Before accepting this solution, one other should be looked at.

The most intuitive way to handle one location into which a processor places its indication is to consider that the processor has its identification assigned to a specific bit in a word. The processor "turns on" its bit. The kernel would know which processors were free by which bits were set. As the kernel assigned processes to processors it could mask the appropriate bit out of the word before assigning the process to the processor. To make this work correctly there would have to be two separate mechanisms to force a process from a processor and to assign a process to a processor. Consider that only one mechanism was needed (as will be seen). Assume that the kernel has just determined that for processor A it should switch from executing process X to process Y. The kernel tells processor A to execute process Y which would force it to stop executing process X as well. Processor A must inform the kernel that it has switched so that the kernel knows that process X is no longer executing. Thus its available bit would have to be set. The kernel knows that this is not really an available bit but a switch bit in this instance and can thus simply clear it. If, however, process X relinquished processor A while the kernel was preparing to force it to
do so, and process $Y$ immediately also relinquishes processor $A$ after being given to it, processor $A$ may set its available bit twice while the kernel assumes that it was only set once. Processor $A$ will never seem to be available, and process $Y$ would never seem to relinquish processor $A$ though the exact opposite is true. Given only an idempotent means of indicating availability, two different mechanisms for forcing a process to relinquish a processor and for assigning a process to a processor are required.

There are two distinct reasons for forcing a process to relinquish a processor. One is that some other process which is deemed to be more important should be given the processor. Another reason may be that the process currently active should be made inactive due to some other reason. For example, if the user of the machine wishes to terminate some task, the processes which are performing that task have to be terminated, whether they are active or not. Rather than implementing two distinct mechanisms one can be used. If a processor is executing a process, and another process is assigned to it, this is sufficient indication that the process currently being executed should relinquish the processor. This still leaves open how to handle a forced relinquish when there is no process to replace the one being forced to become inactive. Honeywell makes a minicomputer, the Level 6, that does process dispatching in the hardware. Each priority level has an enabled bit associated with it. Only an active process at that level can disable that bit. This forced the implementation of Thoth on that machine to have a fake “process” consisting of two instructions. One instruction disabled the bit, and the second jumped back to do it again. When the last process at a priority level became blocked the hardware insisted that it dispatch another process at that level since the bit was still enabled. The fake “process” was used to get the bit to clear. Exactly the same thing is possible here. If there exists a fake “process” which will simply relinquish the processor, to force another process to relinquish a processor that processor only need be assigned a fake process. All that remains now is to decide how to assign a process to a processor.

The kernel needs to know very little about the internals of a process. In fact, if multiple types of processor are to be supported on one machine, it is very difficult for the kernel to know anything about the internals of a process. What it does know is what segments of memory are assigned to each process. This seems the necessary and sufficient information for the kernel to assign a process to a processor. The processor used by the kernel has a FIFO into which the other processors place their identification. The other processors can have FIFOs into which the kernel places the memory description of the process to execute. A processor is in one of only a few states. If it is ACTIVE and the
FIFO becomes non-empty, it goes into the SAVING state. After saving the current state it goes into the AVAILABLE state where it tells the kernel that it is available. From this state it goes into the LOADING state where it takes the contents of the FIFO and uses it to configure its memory management unit and cache. It then goes into the ACTIVE state. The non-emptiness of the FIFO, when in an ACTIVE state is sufficient to indicate that it should switch processes. Details on this are presented in 5.2.6.3.

All processors have a unique location assigned to them. For the kernel processor the location is written to by all other processors, to indicate to the kernel that they are either active or inactive. For all other processors the location is written to by the kernel to assign processes to processors.

Section 3.4 Bus

The bus is the common path over which all accesses take place. There is only one aspect of the bus which can influence the software. This aspect has to do with handling an address which does not match any unit on the bus.

Given that the kernel has properly set the memory mapping limits of all the processes, no process will ever generate an address on the bus which does not correspond to some unit on the bus. It means that, in general, there will never be any invalid addresses on the bus. For normal usage there is no need for any specific solution to be designed to deal with invalid addresses, which implies that the bus need not provide any time-out signals, or such like, to deal with the problem. The "fly in the ointment" is that at some point the kernel must know how much memory is in the machine so that it can assure that it never assigns any process a memory segment that does not exist.

Here again there are two ways to look at the problem. Either the kernel has to be "told" how much memory exists, or it has to find out. If it attempts to find out how much memory there is, then it has to try accessing memory locations until it fails. If this approach is taken, the kernel will access an invalid location. The bus must be designed to deal with invalid addresses. This is excessive since there will be an invalid address only once for every time the machine is powered on. The kernel must be told how much memory there is, either through some hardware specifically designed for this task, such as a special memory location which provides the values of some physical switch settings, or by placing a piece
of memory on the bus which responds in a manner that the kernel can take as implying that it does not exist.

What method is chosen is not important. All that matters is that the bus need not deal with invalid addresses. There is no need for any time-out style of handling the bus. This has implications with respect to failing components. If a memory board fails to the extent that it cannot respond when addressed, the machine will hang, with the address lines of the bus pointing to the offending memory board.

While it appears at first glance a deficiency, it is not. If the memory in question was being used by some trivial process such as one which was working out the first million digits of pi, terminating the process would be a reasonable thing to do if the kernel was given the indication that the memory was faulty. If the memory was used by some critical process, or even the kernel, what to do on failure becomes less obvious. For situations where fault tolerance is required it is far better to have the offending machine "die" in a firm manner rather than to have it linger, confusing the problem. This firm "death" of a machine is also a useful aspect to have when repair is considered. The failed memory board is easily identified since a "finger" is left pointing to it.

From the software view, firm failure is also an advantage. There is no need to produce code to deal with failures of this sort since they cannot be detected. The code not produced does not need to be tested. Given that testing failure mode code is difficult, as the failures seldom happen and are hard to simulate, this removes a large number of difficult problems from the software providers task. Consider the problem of writing the code to deal with failures of memory when the memory that failed holds the code to deal with failures.

Section 3.5 Devices

Any machine will have some number of devices connected to it. In order to be used these devices need to be accessed by processes. These processes have to be created. The kernel has to know the address space of the physical device so that the process controlling it can be given access.

Again this is a "kernel knows" or "kernel is told" situation. It is extremely common for some sort of configuration table to exist which is used to indicate what exact hardware and software components are to be considered. The kernel is less complex since it blindly
can perform its initial process creation task. All is well until changes in the configuration are considered.

If a device fails and has to be removed, or a new device is installed, the configuration table no longer reflects reality, and the table has to change. It is far better if such a configuration table is not static, but dynamic. If the existence of a device makes an entry in the configuration table present, and its non-existence makes the entry non-existent, there is no need to have humans involved with the configuration.

Another aspect of configuration change which has to be addressed is the non-existence of a program to be used by the process which manages the device. When a device is installed not only the hardware has to be physically connected, but the software which deals with the hardware has to be inserted. This leads to the common situation of opening the box and finding the hardware, and a "floppy" disk which has to be inserted, containing programs which have to be run to complete the installation. There is thus a multiple step procedure necessary to introduce a new device. It is common that the physical addition of a device without its associated software causes undesirable things to happen. An interrupt from a device which "does not" exist, cannot be handled. The use of the software without the physical hardware causing undesirable things to happen is also just as common. Certain devices require "busy wait" handling at initialization. If no device is present, no response will be made.

If the physical addition of the device also made the software available, and its removal made the software unavailable, the situation would be simplified. The kernel should have some means of identifying the existence of a device, and the means of identifying what program should be used by the process which controls it. This knowledge cannot be built into the kernel for it would be limited to the situations which were conceived of at its inception. What the kernel "knows" is how to obtain that knowledge.

A solution is to divide the address space which devices inhabit into fixed size pieces. Each device is allowed one of these pieces. The kernel reads the first location of each piece. If the device exists, this location will contain a non-zero value. This value, and some suitable number of following locations, will contain the information necessary to allow the kernel to create the process needed to handle the device. The set of values is exactly the same set as is needed to create any process. Since the kernel knows it is creating a process to handle a device, and it knows which piece of the address space it is currently looking at, it can give the device handler process access to the addresses which the device inhabits. If
there is no device, the kernel has no program to assign to a process, and hence no need to create a process.

For example, assume a two-line serial board exists and is being used. There is a need for more serial lines. An eight-line serial board is purchased, the machine turned off, the old board removed, the new board inserted, and the machine turned on. The kernel steps through the device address space, encounters this new device, and causes the appropriate process to be created, with the ability to address the device's interface locations. That process creates some number of other processes to complete the set of processes which manage the device, then assumes its task of device handling. The machine now has an eight-line serial board, and all the processes necessary to deal with it.

The kernel need not know what a device is. A "thing" exists at some location in the device address space. The contents of "X" locations indicate the values needed to create a process to handle the "thing". That process can do whatever is necessary to make the "thing" a functional component of the machine.

**Summary**

This chapter has discussed aspects of the operating system and the hardware where they interact. It is a short chapter because there is really little interaction between the two if both are designed to interact.

One major point is that the hardware is segment based, with eight segments. Four segments can be modified. Four segments, two modifiable and two not, can be cached. The identification of which segments have which attributes is based on the segment address. The memory is allocated to segments using a buddy scheme. An algorithm has been given which allows efficient compaction with a buddy system.

By identifying a set of features which are provided by the processors, and matching "must have" and "would like" indicators to programs, it has been shown how a multi-processor machine can have processes assigned to processors. Such an assignment even supports heterogeneous processors which belong in non-disjoint sets.

The bus of the machine has the complete physical address space filled. This is done either by actual memory and devices, or "stubs" which complete the address space. This
allows all addresses to be “valid” and thus there is no need for timeout handling on bus accesses.

Devices are integrated into the machine in the form of segments of the physical address space of the machine. Each device provides the program which will support it, as well as the access to allow the support.
Chapter 4
The Operating System

The operating system of a message passing system should be constructed from a large number of small processes which communicate to provide the services needed. This should be the case for many reasons. To tell others that the reasonable way to handle a complex task is by decomposing it into many small communicating processes is fine. To show your belief in this by producing an amorphous single process operating system leads to a lack of trust in your statements. How much confidence would be instilled by a proponent of air travel if he or she used the train? The inherent benefits of process structuring are of great use in constructing an operating system.

One problem does arise if the operating system is constructed of a large collection of communicating processes. In a monitor style operating system there is that amorphous mass that can be pointed to and called "The Operating System", but for a set of communicating processes this is not true. The boundaries of the operating system are fuzzy. The programs used by processes which can be considered part of the operating system can be used outside the operating system. The set of processes which can be identified as belonging to the operating system can change over time. So too can the set of programs. As well, asking any two people to identify what programs and processes constitute the operating system will usually generate two different answers.

The structures of various common styles, or prototypes, of process which are found in and around the operating system will be covered. After this a detailed discussion of the kernel of the operating system will be made, followed by a detailed examination of various groups of processes, roughly collected into commonly identifiable large sections of the operating system.

Section 4.1 Process Prototypes

Processes can be classified as servers or clients. This distinction is not sufficient, for in any large collection of processes there will be some which have the characteristics of both. When a collection of processes work together to provide some service, certain general styles of process become apparent. Four of these, the Owner, Owner-Driver, Administrator, and Courier have previously been identified in [Gentleman 81]. Styles appear due to the fact
that the processes each provide some sort of service, and that service tends to define how the process should operate.

Humans, being what they are, find it easier to conceptualize the differing process styles by analogy to human prototypes and so that will be the approach taken here.

4.1.1 The Owner

A basic style of server can be termed an owner process. The algorithm for an owner process can be seen in figure 4.1.

```plaintext
Repeat
   Receive request
   Service request
   Reply with answer
Forever
```

Figure 4.1 Owner Algorithm

A social analogy which fits well is that of the owner of an older style barber shop with a single chair. The barber waits until a client enters and asks for some service such as a shave or a hair cut. While servicing that client any other clients which arrive have to wait until the barber is finished with the first customer. The barber roughly follows the algorithm seen in figure 4.2.

```plaintext
Repeat
   Wait for customer to sit in chair
   If Shave requested
      Then Shave customer
   Else Cut hair
      Collect money and let customer leave
Forever
```

Figure 4.2 Barber Algorithm

Within a computer system, a typical process of this style is the process which would manage a time-of-day clock. It would roughly follow the algorithm seen in figure 4.3. Such
a process would allow any process to request the current time and could control which processes could set the time. Note that this process forms a typical process in a communicating process model. It is the only process which sets the time, and it would do so on behalf of the process that requests the setting of the time. It hides the physical implementation of the time-of-day clock from all processes which use it. It can provide a uniform definition of what the time of day looks like, and how to change it, no matter what the underlying hardware may be.

```
Repeat
  Receive request
  If Set the time
  Then Apply time in message to hardware
  Else Place hardware time in response
  Reply with the response
Forever
```

Figure 4.3 Time of Day Server Algorithm

4.1.2 The Owner-Driver

A basic style of server can be termed an owner-driver process. The algorithm for an owner-driver process can be seen in figure 4.4.

```
Repeat
  Send for request
  Service request
  Send answer
Forever
```

Figure 4.4 Owner-Driver Algorithm

This style of process is analogous to the owner-driver of a large truck. Many small one person trucking companies have an excessive overhead which can be reduced by working with other owner-drivers. Rather than each dealing directly with customers they consolidate and work through a common office which maintains a staff specifically to support the collection of owner-drivers. The benefits that can accrue reduce the overhead that each would incur individually and help to spread the workload evenly. As an added
benefit to the customers, the number of individual ads in the telephone book is reduced and
the single ad is larger and easier to find. Further, because the workload can be evenly
distributed, should the customer not care which driver is used the chances of earlier delivery
are greater. The owner-driver of a truck would tend to follow the algorithm seen in figure
4.5.

<table>
<thead>
<tr>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ask for work to do</td>
</tr>
<tr>
<td>Make delivery as told</td>
</tr>
<tr>
<td>Report delivery made</td>
</tr>
<tr>
<td>Forever</td>
</tr>
</tbody>
</table>

Figure 4.5 Truck Driver Algorithm

A typical process of this style is a handler of a specific printer when there are a
number of printers available. Rather than directly receiving requests from client processes it
receives them indirectly through an intermediate process with which the client processes
communicate. The client process sends a request for a file to be printed to the intermediate
process which then can pass the request on to an available printer handler. The algorithm
used by the printer handler would tend to look like that in figure 4.6. The system could
function with the client processes sending messages directly to the printer handler, with the
printer handler operating like the owner process seen earlier, but the situation could easily
arise where there were many processes queued for a particular printer, while all the rest
went idle.

<table>
<thead>
<tr>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ask for file to print</td>
</tr>
<tr>
<td>Print file as told</td>
</tr>
<tr>
<td>Report print complete</td>
</tr>
<tr>
<td>Forever</td>
</tr>
</tbody>
</table>

Figure 4.6 Printer Handler Algorithm

4.1.3 The Distributor

A basic style of server can be termed a distributor process. The algorithm for a distributor
process can be seen in figure 4.7. The distributor can be typified by the fact that it never
sends messages to another process, but receives them. It is always available to work for client processes, although the work it does is minimal.

<table>
<thead>
<tr>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Receive request</td>
</tr>
<tr>
<td>If Client Then Queue client</td>
</tr>
<tr>
<td>If Worker Then Queue worker</td>
</tr>
<tr>
<td>If Client queued and worker queued Then</td>
</tr>
<tr>
<td>Reply to worker</td>
</tr>
<tr>
<td>Reply to client</td>
</tr>
<tr>
<td>Forever</td>
</tr>
</tbody>
</table>

**Figure 4.7 Distributor Algorithm**

Working with owner-drivers are the processes which can be typified as distributors. The social analog is that of the dispatcher for a taxi company. Clients of the taxi company make requests for a taxi, and taxi drivers make requests for clients. As clients requests arrive they are queued and as taxi drivers report in, they too are queued. When there is a queued client and a queued taxi driver, the taxi driver is dispatched to the client. Within a computer system the intermediate process between the client processes and the printer handlers tends to be a distributor process. Note that the distributor process does not do any useful processing. It redirects messages from client processes to the printer handler processes that the client processes should have directly sent to. Its usefulness arises from the fact that the client processes cannot be expected to "know" which printer handler it should send to. It provides for the number of printers to change with time. With one printer, the printer handler would be a typical owner process, and all client processes would send to the PRINTER process to have files printed. With the introduction of a second printer the printer handlers would become owner-driver processes and the distributor process would take the place of the PRINTER process. To the client processes nothing would have changed. The addition of yet another printer would just increase, by one, the number of printer handlers acting as owner-drivers.

### 4.1.4 The Administrator

A basic style of server can be termed an administrator process. The algorithm for an administrator process can be seen in figure 4.8. It looks amazingly like that of the
distributor. The administrator can be typified by the fact that it never sends messages, only receives them. It is always available to work for client processes.

```
Repeat
  Receive request
  If Client Then Queue client
  If Worker Then Queue worker
  While Client ready Do Reply to Client
  While Worker ready Do Reply to worker
Forever
```

Figure 4.8 Administrator Algorithm

The social analog of an administrator can again be drawn from the trucking industry. When the number of owner-drivers was small, all the distributor did was redirect client requests to an available truck. As the business grows, the tasks grow as well. Rather than appearing as a simple owner to the client the business starts to show diversity. The client can request express delivery, or normal delivery. As well as moving furniture the company can provide packers who prepare the furniture for shipping, and warehousing for storage. The company now is offering services in its own right, apart from the simple trucking that it originally did.

The computer equivalent can be seen in the administrator of a set of serial lines. As well as acting as an intermediary between client processes and serial line handling processes it can provide other services. The serial line handling processes work in a slightly different way from owner-drivers (see next section), and the administrator takes on some new tasks. It buffers input which has arrived on various lines and can pass this data along to clients which request it. As well, it can provide for such services as timeouts and minimum data sizes on input requests.

A client could request that it be replied to if input arrives, or within a specific period whether or not input has arrived. This is useful for such tasks as terminal emulation. Keeping the display totally up to date with characters that have arrived over the serial line can be expensive. By delaying some of this expense until a few characters can be processed
at a time, the total overhead can be reduced. A good case in point was seen in the first terminal emulator which ran under the Port operating system. At 9600 baud everything worked well and no input was ever lost. When the serial line speed was changed to 1200 baud, input from the serial line was not all captured and characters were lost. This went completely against any intuitive expectations. The basic problem was that, when the speed of the serial line was reduced to 1200 baud, each character was being processed individually through the whole collection of processes used to provide the terminal emulation. Since handling the input and buffering it were more “important” than showing the characters on the screen, the screen handler was of lower priority than the other processes. A speed of 1200 baud was synchronized with the overall speed of data flow through the collection of processes so that no time was left for the screen handling process to remove characters from the buffer, and the buffer overflowed. At 9600 baud the characters appeared to arrive in clumps, and were processed in clumps, reducing the overhead per character, and resulting in no lost characters. By implementing serial line handling so that an input request could specify a minimum count and a timeout, everything functioned smoothly at all baud rates. A minimum count of eight characters allowed the screen handler a chance to run and dispose of the characters in the buffer. The timeout of a tenth of a second allowed the handling of the end of transmission. When input stopped arriving, the input request timed out, any characters that had arrived were passed through, and then the input process of the terminal emulator switched to using a minimum count of one character, and an infinite timeout value. While input was arriving in a stream it was processed in clumps, and, when no input was available, no processor time was used in handling timeouts.

An administrator can provide other services such as statistics gathering. It is essentially an owner process which has expanded to the point where it has other processes working for it in order to provide services to client processes.

4.1.5 The Tradesman

A basic style of server can be termed a tradesman process. The algorithm for a tradesman process can be seen in figure 4.9. In general it appears much like the owner-driver process.

This type of process is analogous to a skilled tradesman. While similar to an owner-driver process, when looked at in detail it is quite different. From the same trucking industry example, the human analogs would be found in the warehouse workers and office
staff. These people are not directly performing any service for clients, but are instead filling supporting roles which allow the company to service its customers.

```
Repeat
  Ask for work
  Perform task
  Give results
Forever
```

Figure 4.9 Tradesman Algorithm

A simple example from a computer system is seen in free space management for a file system. The file system administrator needs to find free blocks of storage when a file is either created or grows in size. The implementation of free space management could be done within the file system administrator, or given to a worker. Placing this task in a tradesman process would simplify the implementation of the file system administrator, and make it more easily understood. As well, being encapsulated within a simple tradesman process, the method of recording which blocks were free can be hidden from all those processes which need not know that level of detail. Changing the tradesman's program from one that used bit maps to one that used pointer-extent pairs would be transparent to the file system administrator, and justifiably so. The tradesman process could choose an appropriate method of recording free space based on the data it was currently working with.

4.1.6 The Receptionist

A basic style of server can be termed a receptionist process. The algorithm for a receptionist process can be seen in figure 4.10. The receptionist process is a worker for an administrator process, like the tradesman process, however its duties are entirely different.

The receptionist process is placed between the client processes and the administrator. It can act as a funnel for client process requests, and as a screening process to simplify the work of the administrator.

The human analog of a receptionist process which acts as a funnel can be seen in a doctor's office. A patient arrives and deals with the receptionist, who has the patient wait until the doctor is free. The receptionist then tells the doctor of the presence of the patient. Without the receptionist the patient would interrupt the doctor to announce his or her
presence. The doctor cannot deal with the patient until finished with some prior arrival and so should not be concerned with the existence of yet another patient.

```
Repeat
  Receive request
  If Valid request
    Then
      Send to boss
    Else
      Result is failure
      Respond with results
  Forever
```

**Figure 4.10 Receptionist Algorithm**

The computer version of the funnel receptionist can be seen in a buffering situation. The process doing the buffering has a limited number of buffers. It has a limited amount of space in which to hold information about processes which are trying to add data to the buffers when all the buffers are full. The introduction of a funnel receptionist between processes attempting to add data to the buffers and the buffering process can simplify the task of the buffering process. If all the buffers are full, and the funnel receptionist sends a request to add more data to the buffers, the buffering process remembers the single request of the funnel receptionist. Any other processes trying to add data to the buffers will block waiting for the receptionist process to receive their requests. The buffering process has to deal with one possible data adder rather than a potentially unlimited number. A similar receptionist can be used to funnel remover's requests, simplifying the buffer process even further. It appears to have one adding process and one removing process. With all buffers full it receives from the removing receptionist, with all buffers empty it receives from the adding receptionist, and in between it receives from either. This solves the problem of what to do with data that cannot be stored, and information about data that cannot be stored.

The human analog of the screening receptionist can be seen in any bank. When a customer applies for a loan the customer does not directly deal with the section of the bank which handles loans. The customer deals with a loans manager. The loans manager does not loan money to the customer. The loans manager checks to see that the customer is acceptable as a borrower of the amount requested, and should that prove true, passes the loan request on to higher authorities within the bank who do lend money. The loans manager has assumed the task of checking that the loan is acceptable. The higher authorities
accept the recommendations of the loan manager and need not check each loan for acceptability in as great a detail.

The computer version of a screening receptionist can be seen in the computerized version of a bank. Deposits to an account specify an amount and account number. This can be directly passed to the process which administers the recording of funds within the bank. Withdrawals from an account need some sort of validation that the withdrawal is acceptable. A screening receptionist process does this checking, and the funds administrator keeps the books correct. A request for a withdraw which comes from the appropriate receptionist is by definition acceptable. A withdrawal request from any other process can be rejected.

4.1.7 The Courier

A basic style of server can be termed a courier process. The algorithm for a courier process can be seen in figure 4.11. The courier process is a very specific style of process with a very specific task to perform.

```
Repeat
  Send for request
  Send request
  Send response
Forever
```

Figure 4.11 Courier Algorithm

One major use of the courier process is as the process to insert so that there are no cycles in the send-receive graph, as mentioned in chapter 2. The other major use of the courier process is as a specialized worker for an administrator process. When one administrator needs to make use of the services offered by another administrator, there are two simple choices. It can either make use of these services directly, or indirectly by the use of a courier.

Consider the situation where a file system is spread over a number of storage drives. One drive is a floppy drive with extremely long seek times. Another is a hard disk drive with very fast seek times. If the file system administrator sends a request to the floppy drive administrator, all file system requests would have to wait until the slow floppy drive request was serviced. If a courier were used, requests for the hard drive could be serviced while the
service of the single floppy drive request was taking place. All other floppy drive requests would have to wait, but the hard drive would be completely available for use.

4.1.8 The Notifier

A basic style of server can be termed a notifier process. The algorithm for a notifier process can be seen in figure 4.12. A notifier is yet another specialized form of worker process. Its task is to wait until a specific event has occurred, and then notify the process it is working for, that the event in question has occurred.

```
Repeat
  Wait for event
  Send notification
Forever
```

Figure 4.12 Notifier Algorithm

A typical notifier is one which waits for a specified amount of time to elapse. This sort of notifier is quite typical in process control applications where sensors have to be sampled and displays updated every so often.

The sensor process would use a notifier to tell it when to read the sensors it manages. Various notifiers could be used to provide various sampling rates for various sensors as required. While waiting for requests from its notifiers, it would be available to service requests for updated sensor values. Processes could send requests asking to be replied to if certain sensors are outside acceptable values. Other processes could send in new acceptable values for certain sensors.

At the same time a display management process would make use of notifiers to inform it of the need to update certain displays. Whether the display management process asks the sensor management process directly for the values or uses a courier is dependent on the exact situation.

Notifiers other than simple timers are possible. For example, a notifier can be used to convey the information that the user has removed a floppy disk from the drive, or that the printer is out of paper. Notifiers are used to remove the need for polling in many situations.
Section 4.2 The Kernel

A message passing operating system is, in general, built around a small kernel which provides the basic level of support for the message passing method chosen. It simulates the "instructions" the hardware designers left out. The kernel could be designed to provide all the operating systems support needed or desired, but this tends to be a poor decision for at least two reasons.

First, there would exist, within one address space, many disconnected activities such as handling the file system and the time of day. In itself this is not an invalid approach, but it does entice the systems programmers into changing these disconnected activities into cunningly connected activities. The kernel would then become slightly more "efficient" and much less understandable and maintainable. Documentation of a piece of code in such a system tends to contain phrases such as, "... don't have to because the ...", and "should not change because ... depends on ...", if such dependences are documented at all.

The second reason for a small kernel is an argument for credibility. It is hard to argue that breaking an application down into separate communicating processes is the correct method to use if the only application provided by the designers as an example is a large monolithic convoluted kernel which defies all the rules they espouse. It is easier to lead than to drive.

Properly done, the kernel of the operating system is almost "just another process". The tasks it performs tend to make it slightly different. For example, the kernel cannot be sent a message since it implements message passing. It is possible to pervert the code so that it appears that sending to the kernel is possible, but all that does is introduce extra code and delays in processing send requests to real processes.

The basic task of the kernel is the support of message passing, but other things must be handled. To support passing messages between processes, there have to be processes. To have processes, there have to be programs that those processes can use. This discussion will start with program management aspects of the kernel, cover process management aspects next, and then cover message passing. It will be assumed for this discussion that there exist processes and programs already. The initialization of the kernel will be covered at the end. Before that point other minor, but equally important aspects of the kernel will be covered.
4.2.1 Shared Segment and Program Management

The kernel has to provide some means of programs being added to the set which can be used by processes. This is a very simple section of the kernel. It has very little to do with the whole concept of programs. Because a loaded program can be shared, it naturally falls within the area of shared memory management. Programs are by far more important than shared data segments. Most of this section will deal with that topic.

Initially some program was created by some means. It was compiled at some time in the past on some machine. Where and when the program was created is not important. This program would usually reside in some file in the file system. To be used by a process this program has to be present in memory. This piece of memory has to be a separate segment. It will be used as the first non-modifiable segment by some process.

It is conceivable that the kernel would perform all the actions necessary to make a program usable as a segment by a process. This would not be advisable for at least two reasons. The first reason is that the basic rule in a good message passing system is, "Hire someone." It is far better to have some other collection of processes deal with the mundane tasks involved and only have the kernel do what it must. This aids the simplicity which makes a system maintainable. The second reason is that assumptions built into the kernel may be wrong.

The original program which is going to be made useful to a process may very well not exist anywhere but in memory. It is reasonable to assume that some process somewhere could cobble together a program which is to be used by another process. This is one of the ideas behind compile-and-go language translation systems. A compiler reads a source description of a program, produces that program in memory, and then has that program executed by a process. If the kernel "knew" it had to load a file, the compiler would have had to "think" of a file name, place the program in it, and then have the kernel load that file into another piece of memory. The compiler would then have to remove that file.

The kernel assumes that the program has, in some manner, appeared in the modifiable shared memory segment of the process which requests that the program be added to the list of loaded programs. This shared memory segment must not be currently shared with any other process. When a process requests the creation of a new loaded program it gives up its modifiable shared memory segment, and in return is given an identifier of that new loaded
program. This identifier has to be provided when a process is to be created using this program.

The kernel needs some means of keeping track of these loaded programs. It has to know where they are, how big they are, and how many processes are currently using them. If no process is using a program, it can be deleted. One other little piece of information is needed. Some means of finding the description of a loaded program from a supplied loaded program identifier has to be provided.

Choosing to change a shared memory segment to a loaded program segment was not an arbitrary decision. Both are segments which can be, potentially, shared by more than one process. Both are classifiable as a segment which must remain in existence as long as there is at least one process which is using it. When the use count goes to zero, the segment can be deleted. The subtle difference is that, for a shared memory segment, this deletion can occur immediately, while for a loaded program this deletion should be delayed. If both shared memory and loaded program segments are recorded in the same data structures then a fifth piece of information has to be provided. This piece of information indicates the difference between the two types of segment.

A loaded program segment can be deleted if it is not used. It would be considered good form to not delete it before any process has had a chance to use it. This could happen when the machine was close to total memory utilization. The memory, currently occupied but not used, would be that which contains loaded programs that were not assigned to a process. If the newly provided program was the last on the list of unused program segments, it would be deleted after all other unused program segments were deleted and enough space was still not available. Reloading a program is always a potential requirement. The situation which would require reloading the program when memory availability is limited is no serious problem.

One should note that other schemes are possible for “locking” a new program segment. For example, marking it is special until it has had one process assigned to it is easy, but then the problem of what to do if the process can not be created because there is not enough memory for the data segment has to be handled. If this were the case then either the program segment is no longer special, or it still is. If it is no longer special then it can be deleted like any other unreferenced segment (leading to the situation noted above.) If it remains special then it will possibly never be deleted if no process ever again requests that it be used for a new process.
Because there should be a reasonably large limit to the number of segments which can be shared the amount of information kept about each should be as small as possible. These five pieces of information can be quite conveniently packed into one 16-bit and two 32-bit values. This gives a total of 80 bits for each segment descriptor. These three values can be seen in figure 4.13.

![Figure 4.13 Shared Segment Descriptor](image)

The location in memory and the size of the segment can fit nicely into one 32-bit word. Because a buddy system of memory allocation is used, the size of the segment can be recorded as the \( \log_2 \) of the size, which would require five bits. Working with a minimum segment size of 32 addressable units is reasonable so the base address of the segment would have the least significant five bits zero in any event. When the hardware is covered in detail in the next chapter, it will be seen that this is essentially how the memory management unit is given the base and limit for a segment, so it is a natural method of storing the two values. The hardware requires use of one more bit so the segment granularity is 64 addressable units and not 32. The least significant bit is used for cache control.

The identification of a segment needs to be stored with the segment descriptor to support changes to the use of that segment descriptor. This is obvious when a loaded program segment is considered. If the segment gets discarded and then reused, the identification must change so that any process which had stored information about that loaded program can be informed that it is no longer loaded. The identification has to be used to quickly find the segment descriptor. Both requirements are easily handled by producing a composite identification. This identification consists of the index to the segment descriptor...
in question, concatenated with a "generation number". If there are 2048 segments, the lower eleven bits contain the segment number, and the upper bits contain a generation number. Whenever a segment is allocated, the generation number is incremented. If reuse of a segment is delayed for as long as possible the probability that an identification has had a chance to become invalid, and then valid again is very slight. With 2,048 segments, with only one free, a twenty bit generation number would guarantee that at least 1,048,576 segments would have to become invalid before the segment identification would appear to be valid again. This is a reasonably large number, and very unlikely. If it is assumed that 1,024 of the 2,048 segments have an extremely long lifetime, there would be 1,024 segment descriptors which would be allocated and reallocated. This would result in up to 1,073,741,824 segment invalidations before cycling occurred. Even in the worst case the probability is not worth considering.

To assure that no unused segment descriptor ever appears to be valid, all that is required is that the stored identification not contain the index to that segment descriptor. When unused, the segment identification can be stored in the base and length word of the descriptor. This allows the segment identification to be changed to invalidate any reference, while maintaining the information necessary to advance the generation number on the next use.

There is one other aspect of identification which has to be recorded. The shared segment can contain a loaded program, or can be a shared segment of data. This difference can be easily expressed as one bit. The final composition of the identification is as shown in figure 4.13.

The use count value can be assumed to always be storable in a 16-bit word. It is a count of the number of processes which are currently using the shared segment. Assuming a limit of 65,536 active processes seems reasonable. For an unused segment descriptor this field is useless, as is the base and limit value. In such a situation, it can then be used as the link field, which "points" to the next free segment descriptor. Initially all segment descriptors are free, so all are linked together.

Given the above description it is easy to deduce the three algorithms used to validate a segment, allocate a segment, and free a segment.
Validation is the most trivial. It consists of checking that the stored identification matches the given identification, as shown in figure 4.14. The MASK should, for efficiency reasons, be a power of two, minus 1.

```
if( given==Shared_ident[given%MASK] ) valid = TRUE;
else valid = FALSE;
```

Figure 4.14 Shared Segment Validation

Allocating a shared segment descriptor is simple. If one is available, it is taken off the head of the queue of free descriptors. The identification has to be given a new generation number, and the appropriate type bit has to be set, as seen in figure 4.15. The positioning of the type bit between the index and generation fields allows the algorithm to avoid concern about the generation number going out of storable range. Incrementing a generation value which is the maximum storable results in a generation number of zero.

```
if( new==Shared_head )
{
    Shared_head = Shared_use[new];
    Shared_ident[new] |= type;
}
return( new );
```

Figure 4.15 Shared Segment Allocation

Deallocating a shared segment requires the saving of the identification in the base and value word, the invalidation of the identification, and the linking of the descriptor to the tail of the queue, as seen in figure 4.16. The identification is saved with the type bit zero to simplify allocation. The identification is made invalid by setting it to zero. It is well known that zero is quite often the value provided by software when it is working incorrectly. A segment identifier of zero has to be treated as always invalid for this reason, making use of the first segment descriptor tricky. If segment descriptor zero was an acceptable segment, the method of producing a new generation number would have to be changed. Complexity could be added to the allocation algorithm to deal with this, but there would be one other effect. The detection of no free descriptors would have to be based on the number free,
rather than on the first free. It is far better to never use the first segment descriptor. Wasting 80 bits of memory for the first segment descriptor will save more than that by simplifying the code to implement the algorithms, and will allow the use of the simpler and faster algorithms.

```
Shared_base[old] = Shared_ident[old] & ~TYPE_BIT;
Shared_ident[old] = 0;
Shared_use[old] = 0;
if( Shared_head == 0 ) Shared_head = old;
else Shared_use[Shared_tail] = old;
Shared_tail = old;
```

Figure 4.16 Shared Segment Deallocation

The initialization of the data structures is simple. For each descriptor, Shared_ident[i] is assigned 0. Shared_base[i] is assigned i. Shared_use[i] is assigned i-1. Shared_head is set to the last valid index. Shared_tail is set to the value 1.

Management of shared segments within the kernel has now been adequately covered. Processes are provided with three operations which deal with shared segments. Two are for general shared segment use, and one is for the creation of loaded program segments.

Using the request diagramed in figure 4.17, a process can create a new shared segment. The size of the segment, and whether it is to be used as a read-only or modifiable segment have to be provided. A segment identifier is returned. Should the operation not be possible, a zero is returned. This request may fail if it is not possible to create a segment of the specified size, or if a shared segment of the specified mode already exists.

```
id = New_Share( size, mode );
```

Figure 4.17 Creating a Shared Segment

Any shared segment can be attached by a process by using the request diagramed in figure 4.18. A segment identifier and whether is it to be used as a read-only or modifiable segment have to be provided. If it is not possible to attach the specified segment, a zero is returned, otherwise the given segment identifier is returned. If a segment identifier with the
value zero is given, the appropriate segment is detached, if one is attached. This is the method used to relinquish access to a shared segment.

\[
id = \text{Access\_Share}( \text{id}, \text{mode} );
\]

Figure 4.18 Accessing a Shared Segment

The final operation available for shared segments is the creation of a loaded program segment from a modifiable shared data segment, diagramed in figure 4.19. If the \text{id} given is the \text{id} of the modifiable shared segment of the process, that segment becomes a loaded program segment and the \text{id} of the new loaded program segment is returned. This \text{id} is the \text{id} of the shared data segment, with the \text{type} bit cleared to indicate that it is now a program segment.

\[
id = \text{Make\_Shared\_a\_Program}( \text{id} );
\]

Figure 4.19 Creating a Loaded Program Segment

No mention of how a new segment of real memory is found has been made here. In much earlier discussions the buddy system was covered in great detail. It is not necessary to repeat it. Loaded program segments can now be created and it is time to turn attention to the handling of processes.

4.2.2 Process Management

There are three process management requests provided by the kernel. A process can be created, or destroyed. While it exists, it may request a change in its priority. These three requests will be looked at before the internals necessary to support them are covered.

As a process makes use of a loaded program, and five to seven other segments, process creation appears to be mostly concerned with memory management issues. This is not the case. Of the four read-only segments, the compiled program segment, the library segment, and the system billboard segment all exist and do not have to be allocated. The shared read-only segment is made accessible by direction of the process in question and is not a concern at process creation time. Of the four modifiable segments two, the data generated at compilation time and the message segment, need to be allocated at creation
time. The other two, the dynamic memory segment and the shared modifiable segment, are created in response to process requests.

Figure 4.20 diagrams the process creation request. The process making the request provides three pieces of information. It specifies the identification of the loaded program segment that the new process is to use, the priority it is to originally have, and the identification of one process. In return it is given the identification of the new process which was created, or zero if the creation was not possible. The need for the loaded program segment identification is obvious. The priority and identification of the other process are specified for more subtle reasons.

\[
\text{pid} = \text{Create}( \text{sid}, \text{priority}, \text{contact} );
\]

**Figure 4.20 Creating a Process**

In general the priority of a process is not that important. In specific cases it can be of overriding importance. For example, it is important that the servers in a system be of higher priority than processor intensive applications. Having all file system activity grind to a halt while the first million digits of pi are computed is not to be desired. In a single processor machine, priorities can be used to guarantee, "... will happen before ... ." A multiple processor machine will not provide this guarantee. Priorities provide a weaker version in such a machine. The phrase, "... will not delay ...", expresses the meaning of priorities in a multiple processor machine. When a collection of processes are used as a coordinated set to perform some task the individual processes are not capable of "knowing" their relative importance to the overall task. The ordering is best placed in one location where a coordinated approach can be taken, leading to the requirement that the process be given a priority at creation time.

In a message passing system, a process is relatively useless unless it can communicate with other processes. It needs to know the process identification of at least one other process. In general the most useful process to know about is the process which initiated the creation. By providing a means of indicating a process with which the new process can communicate, the new process is then able to obtain any other information it may need.

Given the means to create a process there has to be some means to destroy it. Using large numbers of communicating processes necessarily implies the transience of a large proportion of them. Server processes tend to exist for a long duration. Apart from this small
minority, processes tend to be created, perform their task, and terminate. Figure 4.21 diagrams the process destruction request.

```
| pid = Destroy( pid ); |
```

**Figure 4.21 Destroying a Process**

Many processes will terminate themselves. They would provide their own process identification as the argument to the destroy request. Such suicide requests will form a majority of the common requests for process destruction. Other situations require that the process being terminated be other than the process making the request. This is common in situations where the services of a support process are no longer required. The example given much earlier about area filling in a graphics system is a case in point. The other area where termination of another process is needed can be found in error handling. If process A receives an undecipherable message from process B, there is no real way it can correctly respond. The termination of process B is a viable option. To provide a meaningless reply would allow process B to continue on its errant way, confusing other processes. To ignore it would leave a process in existence which would never terminate.

The discussion of process creation stated the need for a process to be given a priority at the time of creation. There is a need to allow the priority of a process to change over time. Figure 4.22 diagrams the priority change request.

```
| priority = Change_Priority( priority ); |
```

**Figure 4.22 Changing Process Priority**

There are, in general, two styles of process. A process can interact with the human user. A process can perform computational intensive operations. An interactive process should be of high priority, so that the person using the machine can expect a reasonable response time. Computational processes should be of lower priority so that they do not interfere with the response of the interactive processes. Many processes have aspects of both styles. This request provides the means for any given process to execute at a priority which is appropriate for the current style.

Just as there was a need for descriptors for shared segments, there is a need for descriptors for processes. Processes need identifiers which have essentially the same
characteristics as shared segment identifiers. Much of what was said about shared segment descriptors holds for process descriptors.

The process identifier is composed of two parts. There is the index into the array of process descriptors, and the generation number. Validating a process identifier is done in exactly the same manner as a segment identifier is validated, as shown in figure 4.14. The only difference is that a process identifier has no type bit.

Given that for a loaded program there will be at least one process which uses it at some time, efficiency in allocating a process descriptor is at least as important as efficiency in allocating a shared segment descriptor. The allocation and deallocation algorithms for process descriptors will again be similar to those in figures 4.15 and 4.16 for shared segments. It should be noted that process descriptors are best manipulated by means of pointers. While they can be manipulated internally by using indexes, as shared segments are, it is more advantageous to use pointers to them for most operations. A process descriptor is nearly always in a linked list of some sort. There is little need to know which descriptor it is, only what it describes, and where it is in the list.

As the rest of the kernel is considered, more fields will become obvious within the process descriptor. Process management indicates the need for a few of them. There is the PID field to hold the current process identifier, the PRIORITY field to hold the priority of the process, a LINK field which will be used to support any lists that have to be maintained, and a list of SEGMENT fields to maintain the segment information for the process.

It was mentioned that the kernel has to create the segment to hold the data generated at compilation time, and to do so it must know the size of that segment. It is assumed by the kernel that the first word of the program segment contains the size of the initial data segment. In general the first part of the program segment is treated by the kernel as a description of the program. Various other words will be mentioned in later sections as they become important.

4.2.3 Communication Management

The major use of the kernel by processes is as the means of communication. The kernel provides four operations to this end. Three of them form the message passing interface. The fourth is used for massive data transfers.
Figure 4.23 diagrams the kernel's operation provided to support sending a message. The process has to provide the identification of the receiver. The three other arguments are used to control the area available for data transfers.

\[
\text{pid = Send( pid, base, length, exposure );}
\]

**Figure 4.23 Sending a Message**

It is possible to send a statement which does not require a reply, as well as a question which does. The kernel must have some way of differentiating the two. The method chosen is to constrain the request field of the message. The request constitutes the header of the message, and the other fields the body. The kernel assumes that the most significant bit of the header indicates the difference between a statement and a question. If the most significant bit is a zero, a statement is being sent, rather than a question. Other methods of differentiation are possible. What is an absolute requirement is that the kernel, the process sending the request and the process receiving the message, agree on which is which. Providing two separate operations for sending statements and sending questions allows the sending process and the kernel to agree on the difference. This would require that the kernel tell the receiver of the difference between the two. Building a mechanism to return two different types of response to a receive request is again possible but more complicated. By constraining the message slightly, all three, the kernel and the two processes, can agree on the difference between a statement and a question. This constraint is not a handicap, but an aid. If process A sends a message to process B, the message must contain some indication of what the message is about. Both processes have to agree on the placement of the request. Without any constraint the request field could be placed anywhere. Attempting to integrate two differing sets of communicating processes would become difficult. As a result the request field would tend to be restricted to some known place in the message by operational constraints. Rather than waiting until programs are defined, it is better to settle the matter.

One further aspect of this arbitrary restriction can be of use. By removing the specification of response or non-response indication from the servers in question, every request sent can be either responded to or not. In an earlier discussion it was mentioned that a close request to the file system need not be replied to. If a process wishes to wait until the close has been completed, it need only turn the "response please" bit on in the request, and it will be blocked until the operation is complete. This appears to complicate the server but the exact opposite is true. The server now need not use the request as an indicator of
whether to reply or not. Whatever the request, if the top bit is on, a reply goes out. At the bottom of the server loop a test and possible reply is made. If no reply is needed, or an early reply has been made and the bit is cleared, then no reply is given at the end of the loop.

The three arguments apart from the identification of the process which is to receive the message, indicate the area exposed for data transfers, which was discussed in 2.1.3.6. The sending process indicates the start of the area, its length, and whether it can be read, written or both. Exposure of the address space of the sending process is under the control of that process.

The sending process will block in various states and for various periods of time. The exact blocking characteristics are related to the actions of the process receiving the message, and whether or not the message is a statement or question. The blocking graph is given in figure 4.27 and further discussion can be found after the other three operations are covered.

Figure 4.24 diagrams the kernel's operation provided to support receiving a message. A single receive operation serves as both a specific and a general receive.

```
pid = Receive( pid );
```

**Figure 4.24 Receiving a Message**

A non-zero argument indicates that the receive is specifically from the named process. A specific receive will return a zero process identifier, if the designated sending process does not exist. If the sending process does not exist when the receive is attempted, the receiving process will not block and will be immediately informed of the non-existence of the sending process. Should the sending process cease to exist after the receive was attempted, the receiving process will be unblocked and will receive a zero process identifier as an indication of the non-existence of the sending process.

A zero argument indicates a general receive operation. A general receive will never return a zero process identifier. The receiving process will remain blocked until any process sends a message to it. The blocking graph is given in figure 4.27 and further discussion can be found after the other two operations are covered.

Figure 4.25 diagrams the kernel's operation provided to support responding to a message. When a process is ready to respond, it composes the response into its message.
area and has the kernel pass this as a response to the sending process specified as the argument to Reply.

```
pid = Reply( pid );
```

Figure 4.25 Replying to a Message

The Reply primitive does not block. If the specified process was blocked waiting for a reply, the given process identifier is returned. Should the specified process not exist, or not be blocked waiting for a reply from the process attempting to reply, a value of zero is returned. A non-zero value should "always" be returned. Failure due to the termination of the process after the message was received, and before the reply is attempted is unlikely. It is even more unlikely that the process specified does exist, but is not waiting for a reply. Such a situation would either imply that there is a problem with the program used by the receiving process, or such a long amount of time has passed that the sending process was destroyed, the process identifiers had cycled completely, and the reply was attempted just when the specified process identifier was again valid. The chances of this are extremely small. The kernel returns a status because the receiving process can be a system server such as the file system, and the inconceivable may well have happened.

Figure 4.26 diagrams the kernel's operation provided to support movement of large amounts of data. The three communication primitives provide a convenient means of moving small amounts of data between processes. For larger amounts of data the data movement primitive is useful. The process requesting the transfer has to specify numerous things. The process pid, has an area length pieces long, starting at offset in the exposed area. The requesting process has a buffer at ptr. The direction argument indicates which is to be read and which written. If successful the process identifier specified will be returned. If unsuccessful a zero value will be returned.

```
pid = Transfer( pid, ptr, length, offset, direction );
```

Figure 4.26 Transferring Data

As with providing a response to a message received, it is conceivable that the transfer of data may fail through no fault of the process attempting to transfer the data. As well as the previously stated reasons having to do with the unexpected termination of the sending process, the sending process itself may have "lied" to the process it sent to. As well as
telling the kernel how large the area exposed was, and how it was exposed, it must tell the
process it is attempting to send to. If the sending process exposes 128 locations for read-
only access and then asks the receiver to write 256 locations, the transfer will fail.

The process specified by \texttt{pid} has to be blocked waiting for a response. No check is
made to assure that \texttt{pid} is either directly or indirectly blocked waiting for a reply from the
process which is requesting the transfer. Situations can arise where the chain of blocking
has been broken early due to the structure of the processes between the one transferring the
data and the other process. This can occur, for example, where one administrator style
process has forwarded a request to another through a courier style process.

The kernel is obliged to check that the area exposed does belong to the process which
is exposing it, and that such exposure is acceptable. The provision of the ability to transfer
large amounts of data between two processes is based solely on arguments of simplicity and
efficiency. Without it, the same task can be accomplished by passing small amounts of data
within enough messages to complete the data movement required. The rules of exposure are
exactly the rules of access that the exposed process itself is restricted to. Whether these rules
are checked by the kernel when the statement of exposure is made, or the transfer of data is
attempted is an open question.

Checking the rules when the exposure is made requires checking once. Delaying the
checking until the transfer is attempted may require multiple checks because the data may be
moved in more than one piece. Efficiency indicates that checking when the exposure is
specified is a better choice.

Now that all the communication operations have been covered it is possible to look at
the blocking graph for message passing. This graph is seen in figure 4.27.

Looking at the transition from state 2 to state 1 when the sender sends a statement, it
can be seen that the sender does not block. This transition, the transition from state 4 to
state 1, and the transition from state 3 to state 1, all make two processes eligible for
execution. In these cases one of them was blocked, and the second not active due to the fact
that it had requested the kernel operation that causes the unblocking. This provides a chance
for the kernel to assign processes to processors in a reasonable manner. Only where a
question is sent are two operations needed by the receiver before the sender is again
unblocked.
Transitions
R -> Receiving process attempts to receive
S -> Sending process sends a statement
Q -> Sending process sends a question
A -> Receiving process provides a reply

States
1 -> Sender and receiver both ready
2 -> Receiver blocked till a send
3 -> Sender blocked for reply, receiver ready
4 -> Sender blocked until a receive
5 -> Sender blocked until a receive

Figure 4.27 Blocking Graph

Message passing introduces the need for yet more fields in a process descriptor. The major contribution is made by the protection restrictions placed on the transfer of data. The base, length and form of exposure describing the area of the sending process have to be recorded. It is wasteful of memory to reserve locations in every process descriptor for these fields. A large number of processes will never expose any area to data transfer, and can only do so while blocked waiting for a reply. To communicate with the kernel a process must place the information about that communications in some locations known to the kernel. The description of the exposed area can be left in this kernel communication area and accessed as needed. The extra restriction that this imposes, is that the area exposed for modification cannot include the kernel communication area.

The storing of important information in the data area of a process is a common feature of both hardware and operating systems. Without careful attention this can result in subtle bugs, and security breaches. For example, if the processor status word of a process is stored on the process stack when a system call is made, and it is possible to modify that location from another process, it is possible to "patch" the security level of a common process into one which has operating system privileges. If the register save area is exposed,
more obscure features can be introduced by modifications of the saved registers. There are strong reasons for restricting the area of exposure in common systems on common machines.

The introduction of subtle bugs into one process by another process is not desirable. This is one of the reasons for providing a process with the means of specifying how and what is exposed. No other process can change any location of the exposed process unless that process has permitted it. If the register save area is exposed, it is because the process which is exposed has specified that the register save area be exposed. Placing the data specifying the exposure in registers places them in the register save area and can be considered safe.

Previously it was seen that the rules of exposure should be checked when the exposure is initially specified. This decision was based on efficiency arguments. Saving the exposure specification in an area which cannot be modified leaves the argument intact. If the exposure specification can be modified this is no longer valid. There appears to be three possible solutions to this problem. Either the exposure specification has to be saved in an area which is guaranteed not to be modified, the exposed area must be guaranteed to not expose the exposure specification, or checking has to be delayed until the transfer of data is attempted.

Providing a guaranteed safe place for the exposure specification implies that this place must be outside the data area of the process. Assuring that the exposure specification cannot be modified implies that normal accessing rules be applied, and that special checking of the exposure specification save area be made. Delaying the check until the transfer implies that multiple transfers will require multiple checks. All three solutions have costs. There is a common idiom that, "You get what you pay for." The truth is that, "You pay for what you get." If the check is made as each transfer is requested, that transfer cannot violate the access rules. The check just before the transfer allows the exposure specification to be changed if it is exposed. Because the initial area of exposure is the responsibility of the process exposed, this is not conceptually a problem. Because most transfers will be completed with one transfer request there is no excess checking. Checking at each transfer usually has little extra cost, and it provides some extra capabilities.

Consider a program which is being debugged. All of its data should be exposed to the process being used to debug it. This data can be stored in four distinct segments. It is not possible to specify one area which covers all the data, and contains only locations which are
valid. The validity of the area exposed is not important. The validity of the area transferred is. A good analogy has to do with automobiles and speed limits. A person is not given a speeding ticket because the car being driven is capable of exceeding the speed limit. The ticket is given for using that capability. A process being debugged can expose all of the potential addresses it has, valid or not, and it is only when an attempt to access them is made, does this validity count.

The preceding, rather long, argument implies that there is no need to save the exposure specification in the kernel's process descriptor for the exposed process. Leaving them in the data segment of the process is adequate. There are other data required by the kernel to support communication and these should be in the process descriptor.

Message passing requires two processes. A message can be passed when process A tries to send a message to process B, and process B tries to receive a message from process A. Because one of the processes will always be blocked waiting for the other the identification of the other process will have to be recorded. Were there only a specific receive capability this would be all that is required. The existence of a general receive capability complicates matters somewhat.

If the receiving process blocks before a send is attempted, all is straight forward. When the sending process attempts to send, it is easy to notice that the receiving process is willing to accept a message from any process and the message passing can proceed. If the sending process blocks first, there is an entirely different situation. When the receiving process attempts to receive, a sending process has to be found, if one exists. If searching is required, a considerable amount of time can be wasted. Most servers use a general receive, and so general receives are by far the most common form of receive used. As well as wasting time, a search does not provide any formal ordering on the senders to processes using a general receive. Unless there are any overriding considerations, a first come first served ordering is desirable from a starvation point of view. A list of processes attempting to send to process B is kept, sorted by order of arrival. When process B attempts to receive from any process, the one at the head of the list would be the one in question. This simplifies and speeds the handling of general receives when the sender blocks before the receiver attempts to receive. It has no effect on the situation where the receiver blocks before the sender, whether for a specific or a general receive. It does have a slight detrimental effect on the handling of specific receives when the sender blocks before the receiver.
When process B attempts to receive specifically from process A, and process A is blocked trying to send to process B, process A will be in the list of processes which contains all processes attempting to send to process B. Process A has to be found in that list, and removed. This would be a serious consideration if an unreasonable length of list was common. Any extra cost in such an unlikely situation will have been more than recovered by providing an inexpensive method of finding sending processes to match general receives. For processes which only receive specifically, the number of senders tends to be quite small. This is the very nature of the situation. When the number of senders is large, the easiest solution is to make use of a general receive.

The process descriptor has two fields to support message passing. One is a BLOCKED__ON field which contains the identification of the "other" process, and the second is a SENDERs field which contains the pointer to the process descriptor of the first process attempting to send to the process in question. The LINK field of the process descriptor can be used to maintain this list.

The justification for a list of sending processes has been made. There are two other potential lists which could be kept. A list of receiving processes is possible, as is a list of processes blocked waiting for a reply. The list of senders was implied because there exists a receive primitive which does not specify the other process. The lack of the corresponding send and reply primitives makes these lists almost superfluous. The termination of a process is what requires the word "almost" in the previous sentence.

When a process terminates any process which is blocked attempting to send to, receive from, or waiting for a reply from, the terminating process must be unblocked. Those processes must be identified. The sending processes are easy to identify since they are on the list of processes blocked attempting to send to the terminating process. The other two sets of processes have no explicitly represented link to the terminating process. All process descriptors have to be scanned to detect those processes which have to be unblocked. Unless all three lists are kept, this scanning has to be performed.

Keeping these lists would speed the termination of a process, but would complicate the task. When a process, which was blocked sending to another process, is terminated, it has to be extracted from the list of the other process. Keeping the two other lists would require this processing if the terminating process was blocked attempting to receive, or waiting for a reply.
The overhead of maintaining these two other lists has to be considered in relation to the benefits gained by more efficient termination. If a process never terminates, the overhead involved in maintaining these lists will not be recovered by the saving gained on its termination. The overhead will have to be charged against the savings from the termination of other processes. The overhead of adding a process to one of these two lists is minimal since no ordering need be applied to them. The more expensive part of the overhead is involved in removing a process from one of these two lists. Assume the only cost is that of identifying the process in the list. If there are \( N \) process descriptors, there would be a saving overall if, during the average interval between the termination of two processes, less than \( N \) removals from one of these two lists was done. The comparison to see if an element in a list is the one in question is not the major cost. Adding a process to the list is at least three times as expensive. It requires reading one link and modifying two links. Removing the process from the list is at least twice as expensive. It requires reading one link and modifying one link. This means that less than \( N/6 \) removals must be done to recover the cost of the overhead. Not all receives would require list manipulation but all replies would. When complete message passing is considered, this multiplies the cost by some amount over unity, making the 6 an underestimate. Considering the fact that servers tend to have a number of worker processes waiting for replies and that the cost of removing one of these workers has to be increased by the position of that worker in the list, and it quickly becomes evident that keeping the other two lists is probably not going to be economical. Combining this possible cost with the definite complexity makes it doubtful if the two lists would be of any benefit.

4.2.4 Time Management

There exists a need for some means of allowing a process to delay for a short period of time. As well, there should be some means of rousing a delaying process. One major use of the delay primitive is as a means of controlling a periodic task. Combined with the primitive for rousing a delaying process, it is useful in support of timeouts in such things as communication protocols.

Figure 4.28 diagrams the kernel's operation provided to support a process delaying for a period of time. The time in microseconds is given as an argument, and the identification of the process which requested the early termination of the delay is returned. If no process requested early termination, a zero is returned.
In applications, such as process control, there is a need to periodically collect information from a set of sensors. Between these times the collection process can delay, allowing other processes access to the processor it was using. Should the required period be short enough it may be preferable if the process remain active, polling, until the next time collection is required. The availability of multiple processors makes this a viable option. The processor being used may not be needed. If the period is longer, even in a multiple processor environment the ability to relinquish the processor for a time is valuable.

It must be noted that there is no guarantee that the process will resume execution after exactly the delay period requested. The existence of other processes of higher priority may well force the period to be longer. An extended delay period can be a major concern in a real-time control situation. There are times when the delay period must not be extended. Having the process change its priority before delaying to a very high priority, and then back to a normal priority after the delay assures that the extended delay period is minimal.

Figure 4.29 diagrams the kernel's operation provided to support the rousing of a delaying process. The identification of the process to wake is given as an argument, and the identification of the process woken up is returned. If the specified process was not delaying at the time the wake up request was made, a zero value is returned.

By using a worker process which delays for a specified period, a protocol which requires timeouts can be implemented. For example, if a response is to be expected within ten seconds, the worker process can be told to delay for ten seconds. If no response arrives within that time, the worker will send a message indicating that it delayed for the given time, and appropriate actions can be taken. Should a response arrive within the required period the worker can be roused, received from, and is then ready to be used for the next timeout.

These two primitives appear to satisfy all the requirements for handling such tasks as timeouts. There is a hidden trap. The worker can be sent off to delay for the required
timeout, and when the event in question happens, woken up. The trap appears if the worker has not had the opportunity to delay before the event and subsequent attempt to wake the worker is made. The trap is avoided in a single processor environment by making the worker a higher priority process. In such a case replying to the worker guarantees that the worker will execute, and delay, before the attempt to wake the worker can be requested. In a multiple processor environment there can be no such assurance. With available processors the worker can be assigned to another processor without forcing the controlling process to relinquish the processor. In such an environment a constructive use of the returned value from the \texttt{Wake\_up} primitive can be made. If a process is roused, all is well. If no processes was roused, this is either because the process has not yet delayed, or because the delay period has elapsed. If the time elapsed is shorter than the delay period, it is reasonable to assume that the process has not yet delayed, and the \texttt{Wake\_up} request should be repeated. A simple loop, repeatedly calling \texttt{Wake\_up} until the process is roused will be sufficient, provided the worker process is of higher priority, and still exists. This loop will, in the worst case, take as long as required for the delaying process to be dispatched, and request a delay. Such a situation is very unlikely, but can, on occasion, happen. If the process in charge should provoke the event before sending the worker off to delay, and lose the use of a processor before replying to the worker, the event in question can have happened before even the reply has been given to the worker.

For a delaying process, the period of the delay must be kept. This need not be in the process descriptor. It is safe to leave it in the data area of the process delaying. The value stored there should be the delay time relative to the delay time of the process before it in the delaying list. This will simplify processing when a delaying process is woken up. It also allows the kernel to keep just the delay time of the process at the head of the list for consideration when dealing with the passage of time.

4.2.5 Name Management

With direct communication the identification of a process is needed if messages are to be sent to it. A process is told the identification of the process which is responsible for its existence. This single identification is sufficient as a start to allow a process to obtain many of the other identifications it may need to perform its task. For example, if a terminal emulation task is to be performed, the set of processes needed for this task will be created, and the identifications of the others in the set can be distributed within the members of the set, so that each knows the identification of any of the other members of the set that it may
need. This suffices for finding the identifications of "friends" but there are other processes, such as servers, that such a scheme is less than optimal for.

Falling back to a human analogy, telephone numbers can be considered. People obtain phone numbers by two distinct methods. Either a number is obtained by personal communication or by looking in a directory. If no public access is desired a phone number may be unlisted and only available by personal communication. A business which desires customers always has its number listed in a directory, which can be consulted when some person wishes to contact it. The default with the telephone system is to be listed in a directory. With a computer system the default is to not be listed, because of the difficulty of conceiving of a unique identification that would be meaningful to others. The processes which are listed are those which carry on a "business", the server processes.

Figure 4.30 diagrams the kernel's operation provided to support name registration. A server process informs the kernel of the "name" it wishes to be publicly known by. Its identification is stored in the publicly accessible table in the specified position. A process can be listed under multiple "names". A process can register under a "name" which is currently registered to another active process.

```
I_am( registered_number );
```

**Figure 4.30 Registering as a Server**

A few words have to be said about the seeming lack of rigour in the maintenance of the list. Multiple registration and assumption of entries at first appears to be wrong.

Multiple registration is a convenient method of allowing one process to assume two or more apparently disjoint tasks. For example, input from the user's keyboard and pointing device, and output to the user's screen appear to be two separate tasks. The processes which administer the two tasks would be listed in two unique entries. If the maintenance of a cursor which is moved by the user's pointing device is considered, a connection between the two can be seen. It may be advantageous to make the two processes one process, and factor out the overhead in the communication of the cursor position from the input process to the output process. Multiple registration supports conceptually disjoint tasks being merged in the implementation, if implementation details indicate that the merging is a more desirable method.
Assumption of entries can have two justifications. The most obvious one is that there is an assumption of duties. Examples of assumption of duties are not very common in the "personal machine on the desk" environment. They more easily flow from server machine situations.

If a "feature" is found in one of the programs used on a personal machine, the stored program can be updated. If the program is part of some application, this tends to be all that is needed, for the new version of the program will be used the next time the application is run. If the program is used by some system server, the situation is different, and the machine will have to be restarted. With a server machine the restarting of the machine can be done but doing so terminates all activities on that machine. If the process currently using the program can be replaced by a process using the improved program, there is no need to interrupt any other service the remote machine is providing. For example, the print service could be replaced without resorting to terminating the file service that the machine was providing. All future requests for printing would go to the new server, and the old server would be ignored.

Another use of assumption of entries is the insertion of monitoring or filtering processes between client processes and server processes. Here the new process does not assume the duties of the old. The old process continues to perform its task. Client processes send their requests to the new "server", which sends them on to the real server after performing its designed task.

These two uses of the assuming of the identity of a server process can be of great benefit. These benefits make a valid case for allowing it. Some concern for security can be expressed. Attacking this security problem would be treating a symptom rather than the disease. Providing the ability for creation of processes which use an untrusted program is the real security problem.

In a server machine, providing for the creation of processes by any user is a facility which should not be supported. For a personal machine any valid process has to be created. In general, it matters little if some other process either takes over the role of a server, or filters messages to it. One example where it does matter is an encryption server. Placing a monitoring process between the encryption server and client processes should be invalid. A simple solution to this is to have the encryption server check to see if it is listed as the encryption server every time it receives a message. If any monitoring process has been
introduced, this simple test will detect it. Client processes should request a confirmation that the encryption server is authentic. Replacing the server would be detectable.

Implementation of this primitive requires an array of process identification numbers. This provides a bottom level name server which deals adequately with most name server requirements. Should a more elaborate name server be required, a process can provide such a service, and use this primitive to register as the "name server".

4.2.6 Dynamic Data Management

In general the sizes of the segments of memory used by a process do not change during the lifetime of the process. The only segment which can change is that which is dedicated to dynamically allocated memory.

Figure 4.31 diagrams the kernel's operation provided to support changes in the size of the dynamic data segment. The process specifies the desired dynamic data segment size, and is returned the real dynamic data size. The size returned may not be the size specified for one of two reasons.

\[
\text{size} = \text{Set\_Data\_Size}(\text{size});
\]

Figure 4.31 Changing Dynamic Data Size

The returned size may be greater than the specified size due to the fact that a buddy system of memory allocation is used by the kernel. If the specified size is not a power of two, it is increased to the closest power of two. In such a case the real size is returned so that the program in question may be aware of the situation.

The returned size may be smaller than the specified size if memory utilization is extremely high. In such a case, at least an indication of the fact that the requested memory was not allocated has to be returned. If a process requests an amount of memory, and that amount cannot be granted, the kernel has one of two options. It can either allocate a smaller amount of memory, or it can refuse. Only if the requested size is more than twice the previously allocated size, due to the buddy system of allocation, will a smaller amount of memory be possible. In all other cases, if the amount requested cannot be given, the next smallest amount is the amount already allocated. The probability of receiving a request for
more than twice the amount already allocated depends on the implementation of the internal memory management within the program of the process making the request.

In common systems, when the amount of dynamic memory allocated to a process has to be increased, the lowest level of the implementation of the internal memory management tends to ask for large amounts so that repeated expensive system calls to obtain small amounts of memory need not be made. It is typical to find that allocated memory is asked for by 16 to 32 thousand addressable units at a time. Using a buddy scheme within the kernel makes allocating more memory expensive only if the current size of the segment is large, and it has to be copied. It makes little sense to ask for more than needed. If the amount currently allocated is small, and the amount needed is small, the expense of asking is minimal. If the amount currently allocated is large, and the amount needed is small, a large amount will be given anyway. If the amount needed cannot be allocated there is little sense in allocating only part of it, whether the current size is small or large. To go to the possibly large expense of shuffling memory about to partially satisfy a request from a process, only to have that process immediately terminate because of lack of memory seems to be a waste. The final situation is that if the requested amount cannot be provided, the size returned is the current size before the request was made.

4.2.7 Event Management

Some provision has to be made for interrupts. One of the greater advances in computing was the introduction of interrupts. Interrupts provide the ability to support multi-processing. They, unfortunately, introduced timing problems. Despite their faults, interrupts are better than the alternative of polling.

The basic feature of an interrupt is that it announces that some asynchronous event has happened. Figure 4.32 diagrams the kernel's operation provided to support waiting for asynchronous events.

```
Wait_For_Event( event_number );
```

Figure 4.32 Waiting for an Event

An asynchronous event caused by an interrupt is similar to an event caused by a processor becoming free. How the kernel will find out about the occurrence of the
asynchronous event will be covered with the hardware. What is worth discussing here is the handling of the event by the kernel.

When a process must wait for an event, it uses the request diagramed above to inform the kernel. The kernel records the process as waiting for the event by recording it in a table indexed by the event number. One process at a time may wait for any given event. When the kernel is informed of the happening of an event a flag is set to indicate this fact. The number of times an event has happened is not recorded, only the fact that it has happened at least once.

When an event occurs either a process is waiting for it, or not. If a process is waiting that process becomes ready to execute, and the event is not recorded. If no process is waiting, the occurrence of the event is recorded. When a process attempts to wait for an event the event has either occurred or not. If it has, the process remains ready to execute and the event flag is cleared. If it has not, the process becomes blocked and is recorded as waiting for the specified event. There is no "timeout" facility for events. If a process attempts to wait for an event, it will wait until that event does happen, or the process is terminated.

An event either has a possibility of not happening, or it is certain to happen. Some events, such as waiting for input from a serial line, can have no fixed upper bound on the time until the next event. Others, such as disk operation completion events have a reasonable upper bound, but should they not happen, there is a clear indication of hardware failure. Dealing with the failure of hardware devices is outside of the scope of the kernel. If failure detection is considered useful, a properly organized set of processes can be used to manage the device, with appropriate timeouts provided by worker processes.

Two questions can be raised at this point. The first is, "Why not deal with events by message passing with hardware 'processes'?" The second is, "If waits are implemented, why not implement signals?" These will be addressed in that order.

A common suggestion is to communicate with hardware "processes" by using the message passing facilities used to communicate with software processes. This would provide a uniform communication method, integrating and simplifying everything. This suggestion is dismissed for two reasons.
If message passing is used to communicate with hardware "processes", the implementation of the message passing primitives must detect the difference between a hardware process and a software process. The internal handling of these two situations is completely different. Apart from the extra complexity involved, this integration will slow message passing between software processes due to the extra testing needed to detect the difference. Some set of "process identifiers" have to be reserved for hardware "processes". This will complicate the process creation task. Other minor problems could be mentioned but the most telling of all is that the kernel must know how to deal with the hardware. To "send" a message, for example, to the disk controller, the kernel must know how to interpret the contents of the message, and correctly address the disk controller. It must know whether the hardware process will provide a reply (interrupt) or not. It must know all about every device. Building all of this information into the kernel would result in a large kernel which was specifically tailored to one set of hardware devices. If this route was followed it would make sense to build the complete device handling into the kernel. It is a short step to a monolithic operating system. If the kernel is not involved in the dealings of the devices, it is possible to dynamically change the handling of the devices by dynamically changing the programs used by the individual device handlers.

Another good reason for not supporting hardware processes is that it is not necessary. To properly deal with the hardware some number of processes should exist to manage access to the devices. To most processes in the machine, communication with the "hardware" already is done by message passing. Only at the bottom level of these server groups does the real hardware get addressed. At this point communication with the hardware must be assumed since the details of the hardware are very important. Using the message passing paradigm here is unreasonable. The process knows full well what values have to be placed in which device control locations. It knows exactly what the effect of its actions will be. Introducing one more layer seems redundant.

Overall, simulating the hardware in the kernel as a set of fake processes is not desirable. It would tie the kernel to a set of specific hardware, introduce needless complexity in the processes which deal with the hardware, and impact all communications between all processes.

Given that a wait primitive has been implemented there can be an argument made for a signal primitive as well. A facility for two processes to "rendezvous" without exchanging messages would exist. Providing a second method of implementing a rendezvous, is
superfluous, and would complicate matters. Apart from complexity arguments, there would have to be a distinction between hardware and software waits and signals.

Either of the two generalizations, hardware processes or software signals, are "Swiss Army Knife" approaches, where specific tools are more appropriate.

This completes the full description of primitives implemented by the kernel. The process descriptor can now be considered.

### 4.2.8 Process Descriptor

Each of the primitives of the kernel requires some information about the process or processes in question. These pieces of information are, in general, held in a descriptor for the process. Six of these fields have been mentioned in passing throughout this section. There was the PID field which contains the identification of the process, the PRIORITY field which contains the priority of the process, the LINK field which is used to maintain lists of processes, the SEGMENTS field which contains the eight descriptions of the segments assigned to the process, the BLOCKED_ON field which contains a pointer to the descriptor of the process that this process is currently blocked on, and the SENDERS field which points to the list of processes blocked sending to the process in question. The other field that is needed is a STATE field to maintain an indication of the status of the process.

A process can be in one of a number of states. With respect to message passing it can be SEND_BLOCKED, RECEIVE_BLOCKED, or REPLY_BLOCKED. If it is waiting for a period of time, it is TIME_BLOCKED. If it is waiting for an event, it is EVENT_BLOCKED. If the termination of the process has been requested, but this cannot be acted on immediately (as discussed later), it is in a CONDEMMED state. Failing to be in any of these states, a process must be READY, or ASSIGNED. A process which is currently assigned to a processor is ASSIGNED. A process is READY if it cannot be ASSIGNED because of a lack of processors.

Other fields in the process descriptor may be there for efficiency reasons. One which is very useful is the LAST_PROCESSOR field. This field contains the identification of the last processor to which this process was assigned. If the process becomes READY and can be ASSIGNED, this field is useful. If process A was last assigned to processor B, and processor B was last assigned process A, process A should be assigned to processor...
B. The contents of memory that were cached by processor B will be valid for process A, and thus they should be mated.

Other fields, such as DELAY_TIME, EVENT_NUMBER, etc., are possible if needed. Their utility can only be discovered after implementation of both the hardware and software. Proper monitoring will show any areas which could benefit by such fields. A correct decision on their existence rests on information about their advantages and their overhead.

4.2.9 Process Dispatching

There are two aspects to process dispatching in a multiple processor machine. The introduction of a second processor introduces great changes from a single processor machine. In a single processor machine, when the kernel is executing, no other process is. All processes can be manipulated by the kernel. If a process is to be terminated for example, it is quiescent and the operation can be easily done. For a multiple processor machine the process to be terminated may well be currently executing on one of the other processors, and has to be "called back" and made quiescent before it can be terminated. When a process is to be dispatched on a single processor machine this is easily accomplished since the kernel can arrange to have the process given the processor when the kernel has completed its operations. The multiple processor machine complicates this because the processor to which the new process is to be given access may be currently in use by another process, and that second process has to be "called back". This is where the discussion of process dispatching should start.

For this discussion a little understanding of how the individual processors are assigned processes is needed. Process dispatching will have to be delayed for a short period while the background is briefly covered. A fuller description can be found in the next chapter.

Initially a processor has no process assigned to it. It is waiting for a process to "appear" in its port. When a process does so, that processor will execute the program of that process. The process in question can be said to have been assigned to that processor. The process remains assigned to that processor until one of two events occur.
The process can voluntarily relinquish the processor, in which case the processor goes back to waiting for another process to "appear" in its port. This event will happen whenever the process requires that the kernel perform an operation.

The second event is the appearance of another process in the processor's port. This forces the currently assigned process to implicitly relinquish the processor. This is the means by which a process can be "called back". In order to force a process to relinquish a processor, another process has to be assigned to that processor. The processor will "notice" that another process has been assigned at the termination of the currently executing instruction. This, and the previous transition can be seen in figure 4.33.

![Figure 4.33 State Changes of a Processor](image)

This has been a rather sketchy description of the workings of an individual processor, but will suffice for now. What must be noted is that the time until the termination of the currently executing instruction can be very long. For example, if a Lisp processor is implemented, which has garbage collection hidden from the processor "for simplicity", the time until the next instruction is started can be of a long duration. This implies that, in general, the kernel cannot, "call back the process and then ...", but rather has to, "request that the process relinquish the processor", and when that has happened, "now the process can finally be ...". The kernel is in charge of what is happening within the full machine, but is not in control.

Returning to the dispatching of processes to processors, the kernel must be aware of the state of each processor. The processor is either in a Waiting, Working, or Switching state.

If process A is to be given to processor B, and processor B is in either a Waiting or Working state, the kernel can place the process in processor B's port. If the state was Waiting, processor B will immediately begin executing the program of process A and will
be in a **Working** state. If the state was **Working**, processor B will begin executing the program of process A at some time in the future, and will be in a **Switching** state.

If processor B was in a **Switching** state, the kernel cannot give process A to processor B. At some point in the future processor B will "notice" and go into a **Working** state. At this point the kernel is again able to make use of processor B. Processors in a **Switching** state are not available for the use of other processes and cannot be considered when attempting to assign a process to a processor.

It was mentioned that a process in a **Working** processor is not directly under the control of the kernel. The kernel can initiate the "calling back" of that process. For the case of a processor in a **Switching** state, the process it is currently executing will eventually come under the control of the kernel. The process which was given to that processor, and which changed the processor state from **Working** to **Switching** is a different case. It is not under the control of the kernel, and it cannot immediately be "called back". If the process is blocked for any reason, it is not a candidate for dispatching and can be ignored for the rest of this discussion. This results in any interesting process being in one of three distinct sets.

A process may be capable of being given to a processor, but currently not assigned to any processor. Such a process is in the **Ready** set. A process may be currently assigned to a **Working** processor. Such a process can be forced to relinquish its processor if needed, and so is in the **Controllable** set. A process may be currently assigned to a processor which is in a **Switching** state. Such a process is totally beyond any control and can be assigned to the **Untouchable** set. The processes which can be considered when discussing process dispatching are those in either the **Ready** or **Controllable** sets.

Dispatching a process involves two distinct steps. First it must be determined that the process is to be dispatched, and then the processor to which it is to be assigned must be chosen. If the processor to which it is assigned was in a **Waiting** state, the process moves from the **Ready** to **Controllable** set. If the processor was in a **Working** state, the process dispatched moves to the **Untouchable** set, as does the **Controllable** process which was assigned to that processor. The final aim of dispatching is to reach a state where, if a process is in the **Controllable** set, there exist no processes in the **Ready** set which are of higher priority.
The first step in dispatching is to identify which processes are to be dispatched, and which processors they are to be assigned to. The algorithm for this is straightforward.

Start with the list $\mathbf{RP}$ containing the Ready set of processes. The ordering of this list is from highest priority to lowest priority. The list $\mathbf{CP}$ will contain the Controllable set of processes. This list is again sorted by priority from highest to lowest. The set $\mathbf{Fp}$ will contain the set of processors which are in a Waiting state. The set $\mathbf{Wp}$ will contain the set of processors which are in a Working state. The set $\mathbf{DP}$ (processes to dispatch) will contain the empty set. The set $\mathbf{Up}$ (processors to assign processes to) will contain the empty set. Assume the existence of two functions $\text{First}(x)$ and $\text{Last}(x)$ which remove and return the first and last processes from a list respectively. Assume the existence of a function $\text{Ap}(x)$ which returns the identification of the processor on which the process $x$ is currently executing. This algorithm requires the identification of a processor, $a$, from the set $\mathbf{Fp}$. At this point it can be assumed that any Waiting processor will do. The first part of the algorithm is:

\[
\text{while } A = \text{First}(\mathbf{RP}) \text{ and } a \in \mathbf{Fp} \\
\quad \mathbf{PD} = \mathbf{PD} + A \\
\quad \mathbf{Fp} = \mathbf{Fp} - a \\
\quad \mathbf{Up} = \mathbf{Up} + a
\]

The above section terminates if either $\mathbf{RP}$ is the empty list, or $\mathbf{Fp}$ is the empty set. If $\mathbf{RP}$ is empty, the full algorithm can terminate since all processes which can be dispatched have been identified, and all processors which are to be used have been identified.

Should $\mathbf{RP}$ not be empty, the second section of the algorithm must be invoked. This involves forcing processes in the $\mathbf{CP}$ list to move to the Untouchable set by assigning processes in $\mathbf{RP}$ to the processors that are in use by the chosen processes in $\mathbf{CP}$. The identification of two specific processes, one in $\mathbf{RP}$ and one in $\mathbf{CP}$, is required. A further function, $\text{Priority}(x)$, is assumed to return the priority of the process $x$.

\[
\text{while } A = \text{First}(\mathbf{RP}) \text{ and } B = \text{Last}(\mathbf{CP}) \text{ and } \text{Priority}(A) > \text{Priority}(B) \\
\quad \mathbf{DP} = \mathbf{DP} + A \\
\quad \mathbf{UP} = \mathbf{UP} + \text{Ap}(B)
\]

This second section of the algorithm reduces either $\mathbf{RP}$ or $\mathbf{CP}$ to the empty list, or terminates when there does not exist a process in $\mathbf{CP}$ which is of lower priority than any
process in RP. At this point there is a set of processes which must be dispatched, and a set of processors to which they will be assigned. As these processes are assigned the processes taken from CP will be forced from their respective processors.

Mention has been made of assigning processes to processors. For a single processor machine this can be relatively easy. Only one process can be assigned at any one time. There is, essentially, one list of ready processes. This list is ordered using some priority scheme. The process at the head of the list is assigned the processor. For a multiple processor machine the choice must be made of which process to assign to which processor. First the homogeneous processor situation will be covered, progressing in steps until the heterogeneous processor situation is covered.

The homogeneous processor situation is not as simple as it would first appear. There exist three sets of processors, one set for each state that a processor can be in. Nothing can be done with processors which are in the Switching state, so there are two sets of interest, the Waiting set and the Working set. These two sets can be further divided. This division is not based on any aspect of the processor itself, but rather by aspects of the process considered for assignment. These two aspects are the last processor a process was assigned to, and the program that the process is using.

In the previous chapter it was seen that four of the eight possible segments accessible to a process can be considered as candidates for caching. One of these can always be maintained but the other three are specific to the program, and process using that program. If a process can be reassigned to exactly the same processor it had previously, and no other process has used that processor since the process in question did, the contents of the cache memory are valid, and can be reused. Such a match can be termed MINE. This match of process and processor is more efficient than any other, for it will avoid reloading the cache. Of these three segments, one is the compilation time code segment. Should some processor have been previously assigned a process which was using the same program as the process in question, this code segment need not be reloaded. Such a match can be termed OURS. All other processors require three segments to be reloaded. Such a match can be termed THEIRS.
It can be seen that, for each process, there are five sets of processors. Ranked in descending order they are:

- Waiting/MINE
- Waiting/OURS
- Working/OURS
- Waiting/THEIRS
- Working/THEIRS

If the cost of reloading a cache segment is considered independent of which segment it is, these can be given costs of 0, 2, 2, 3 and 3 respectively. There is the extra overhead involved in handling a displaced process. The processors which are in the Working state will have to be assigned some process, so preferring a Waiting to a Working state processor when the costs of segment reloading are similar is an advantage.

While it is generally accepted that the highest priority process should be assigned before any others, this is not necessarily the best solution. Consider the situation where a low priority process A creates a higher priority process B. Assume that there was one Waiting processor a as well as the processor b that the low priority process was using. When the two processes are being assigned, both processors are in the Waiting state. The cost of assigning the two processes to the two processors is either:

\[\text{Cost}(B\cdot b) + \text{Cost}(A\cdot a) = 3 + 3 = 6\]

or

\[\text{Cost}(A\cdot b) + \text{Cost}(B\cdot a) = 0 + 3 = 3\]

where the cost function is computed from the list of costs given above. The first formula reflects assigning the processes to processors by priority, with process B just happening to be assigned to processor b. The second formula reflects the cost, if it is noted that assigning process A would not deny process B a processor, and would result in a Waiting/MINE assignment for process A. The cost of assigning processes in a blind "highest first" manner can be twice the cost of a more intelligent solution. Even if the two processes used the same program the costs of the two methods would be 5 and 3 which, while not a factor of two, is a major difference.

It may at first appear that the way to assure that the cost of the assignment of processes to processors is minimal is to try all possible permutations and pick the least expensive. A simple five pass solution exists. Since each processor can be assigned to one
of five sets which are specific to the process being assigned, it is easy to assign all
processes which have Waiting/MINE processors, then Waiting/OURS etc. Rather than
classifying the processors into five sets, they can be classified into three, based on the
segment reload count. If there are two processes using the same program, and one Waiting
and one Working processor in the OURS sets, one of the processes is going to get the
Waiting processor and one the Working processor. It matters little which is which. A
simple three passes over the set of processes to be dispatched is sufficient to assign
processes to processors. This could be expensive if the number of processes is very large.

In general, there is little chance that there would be many READY processes to be
assigned to processors. Of the fifteen operations the kernel can perform, seven return the
requesting process back to the processor it came from. Three, Create, Reply and
Wake_up, will make the requesting process and one other eligible for assignment. The
Send and Receive requests can make from zero to two processes eligible for assignment,
depending on the type of Send involved, and the state of the other process. The
Wait_for_event request will either return the requestor to its processor, or block the
requesting process. When the event does occur this will make one process eligible for
assignment.

The two requests which can cause more than two processes to become eligible for
assignment are the Delay and Destroy requests. The Delay request removes a process
from the available list, but when the appropriate time comes there may be any number of
processes which all are again READY. The Destroy request can make any number of
processes available. If the process being destroyed is the requestor, the number may be
zero. If a large number of processes were blocked attempting to communicate with the
destroyed process, the number can be large.

Processes which delay are usually worker processes which will immediately send to
the process they are working for when their delay terminates. Attempting to do a perfect
assignment of process to processors in the case where there is a large number of processes
Ready due to the completion of a delay is not worth the effort. In a short period of time the
situation will have changed dramatically.

Processes which become Ready due to the termination of a process are also in the
same situation. The termination of a process is an exceptional condition, and the set of
Ready processes is bound to change radically soon after the processes are assigned to
processors.
It is obvious that the vast majority of the times that the algorithm is required, such a small number of processes and processors is involved that a reasonable assignment is possible with little overhead. For those few cases where the number of processes and processors is large, matching by the MINE, OURS, THEIRS sets will result in a minimum of disruption to the current set of processes, which are likely to regain their assignments in short order.

For heterogeneous multiple processors the situation is slightly more complex. If the set of processor types is disjoint, the above algorithm for process and processor identification is repeated for each type of processor.

The strict rule that if two processes, A and B are capable of execution, and A is of higher priority than B, B will not be executing unless A is executing, is attainable in a single processor machine. In both the homogeneous and heterogeneous situations this rule does have to be relaxed. The existence of processes, executing concurrently with the kernel of the system and which have to be called back, means that, for short periods of time, there will exist low priority processes which are executing but should not be. Heterogeneous machines have to relax the rule even further since there may be a free processor of a type which can only be used by a very low priority process, while higher priority processes have to wait for their type of processor to become free.

The heterogeneous processor situation was earlier dismissed rather lightly. The order in which the types of processors are handled is not important. The twist to the situation which can cause problems is if a given processor can be of more than one type. This sounds impossible, but is not.

Consider a machine with X processors of type ABC. A specific type of program fails to execute correctly and the fault is traced to the fact that the ABC processors do not perform the squiggle operation correctly for all operands. A complaint is filed and a new version of the processor is created, tested and marketed. Unfortunately the ABC' processor is in limited supply since all owners of ABC processors are attempting to update their machines. Of the X processors, Z of them can be replaced with ABC' processors, leaving X-Z older ABC type processors. There are A different programs on the machine. Of these, B will only work correctly on the ABC' processors, but A-B will work correctly on either type. If ABC and ABC' processors are considered to be distinct processor types, there will be times when some of the processes which are READY, but work on either processor
type, will not be assigned to processors since only ABC' processors are available. All programs could be changed to only run on ABC' processors, but that would imply that the X-Z processors of the old type might as well be thrown away since they would never get used. If the ABC' processors are considered a sub-type of the ABC processors, processes using programs which will work on either can be assigned to either, and only those processes using programs which require the new ABC' processors need be restricted in the processors available. While an enviable goal, it does complicate matters greatly.

Consider a situation with two processors, a and b, and two processes, A and B. Processor a is an ABC type while b is an ABC'. Process A requires an ABC' processor, and B will work on both. If both processors are available and process B happens to be assigned to processor b, process A cannot be assigned. It would seem that processes restricted to a sub-type of processor should be considered first.

This is a simple solution but ignores the priority of the processes. Consider the situation where only processor b is available, but process B happens to be of higher priority than process A. If A is considered first, because it uses a sub-type processor, B will not be assigned even though it is of higher priority.

In practice the selection of which processes to assign, and the processors to assign to, is not done by processor type, but by priority of process. For each process considered the set of acceptable processors is extracted from the set of available processors. This set of acceptable processors is a list of processors, ordered by the type of the processor. The previously given algorithm did not consider this ordering. The process and processor selected are the first of both lists. To deal with sub-types of processors, all that is necessary is to assure that sub-type processors appear in the list of acceptable processors after the general processors. This will, if possible, leave the sub-type processors for processes that may need them. At this point, which processors to use have been selected, but the actual assignment of processes to processors is not yet done.

After the processes and processors are identified, this ordering is reversed. Processes which require sub-types of processor are assigned first. This assures that the processes which will work with a more general set of processors will not be assigned to the sub-type processors until the more restricted processes have been handled.

This variation of the algorithm handles sub-types of processors, and can handle the "it would be nice" situations. In the previous chapter, it was mentioned that a program can
specify what type of processor it could be used on, and specify what sub-type of processor it would be "nice" if it ran on. For example, a program which performed many floating point operations would be best executed on a processor which had a floating point co-processor.

The sorted list of acceptable processors would be ordered by closest match. This would assure that the "best" processor was added to the set of processors to be assigned to. When it comes to the assignment of processes to processors, this ordering is again applied within the, this time reversed, ordering of processor types.

There is no guarantee that this modified algorithm would produce the optimum match of process to processor. Pathological cases can exist where a different match would be better. Consider the situation where one restricted process and two unrestricted processes exist, and two sub-type processors with one general processor. If the restricted process is assigned to the sub-type processor which best matches one of the unrestricted processes, a sub-optimal match will have been performed. The only solution to guarantee an optimal match involves a factorial algorithm. Given \( X \) processes and \( Y \) processors, an optimal match can be found in \( O(\text{Min}(X, Y)!) \) which may well be very expensive. Such an algorithm is not cost effective since such an optimal match is in fact a guess. If two programs indicate that they would best be executed on a processor with a floating point co-processor, and only one such processor exists, and the process assigned to that processor does not happen to use any floating point operations before it again makes a kernel request, but the other process does, the chosen matching will have been the wrong one. Given the limited information available to the kernel, the algorithm described here provides an acceptable matching, at little cost.

This straightforward algorithm can be used to match processes to processors. It deals with heterogeneous processors of overlapping types. It caters for cache handling costs. The necessity to choose the "best" processor forces an \( O(N*M) \) algorithm when \( N \) processes and \( M \) processors are to be considered. In the vast majority of cases the values of \( N \) and \( M \) are less than three, making it reasonably efficient. Should the values be larger then the algorithm, while taking longer, is likely to save far more time by its better matching than is possible by a "faster" algorithm.
Section 4.3 Program Management

The kernel manages processes by means of the Create and Destroy requests. It manages programs in a less direct manner. The Make_Shared_a_Program request causes a segment to be recorded as containing a program. The identifier of that segment has to be provided when a Create request is made. The Create will either succeed if the segment exists, or fail if the kernel deleted the segment because the memory it occupied was required for other uses. Direct use of the kernel for the creation of processes is not of great utility to the average process attempting to create another.

The requirement that Create be given a segment identifier, which will be determined when the program is loaded, makes it difficult for programmers to correctly implement the code to create a process. As well, the possibility that the segment may no longer exist would require that the process which is attempting to have a process created must deal with a temporary failure. If the Create fails, it must have the program reloaded, and attempt the Create again.

A desirable situation from the average programmers point of view would be one where the name of the program that the process should use be fixed, and that all details about whether or not the program is currently loaded be taken care of by some other process. Here we introduce the first server in the system.

The Program_manager is a process which provides the services needed by processes, to create yet other processes. To have another process created, the process sends a message requesting the creation of a new process, and provides the path name of the file which contains the program the new process is to use. In the reply to that message will be the identification of the new process. The translation from path name to segment identification, and any loading of the program needed, will be handled by the Program_manager.

The translation is a simple process. A list of loaded program names and segment numbers is kept by the Program_manager, and when a create request is made to it, it finds the name in the list and extracts the associated segment identification. Should the name not be found in the list, the program will have to be loaded.
If the name was found in the list, and the kernel can create a process using the associated segment identification, all is well and the process which sent the message to the Program_manager can be replied to with the identification of the new process.

Even if the name is found in the list, the program may have to be loaded. If the kernel had to delete the segment to reuse the memory it occupied, the attempted creation of a process using that segment will fail. In this situation, the Program_manager must remove the name and segment identification from the list, and continue as if the name had not been found. It should be noted that, because the data area of the kernel is mapped into the address space of all processes, including the Program_manager, the checking that the segment is no longer valid need not be done by blindly attempting to create a process with that segment identifier.

The Program_manager can look into the kernel’s structures to see if the segment identifier is invalid. The check will not assure that the segment identifier is valid, only that it is invalid. As with all information which is stored about situations not totally under the control of the process using that information, there are few YES/NO answers available. In general only one of YES or NO is guarantied. The other has to be prefixed with WAS. In this particular case the Program_manager can either find that the segment identifier is NOT VALID or that it WAS VALID when it looked. The creation of a process cannot be guarantied just because the segment identifier appeared to be valid. It would seem that checking before attempting to create the process is of no use. This is not the case. The overhead of the check is very small compared to the overhead involved in even a failed attempt to create a process. In either event, as was previously stated, the failure to create the requested process due to the loss of the loaded program results in the name being removed from the list maintained by the Program_manager.

If the name is not found in the list, the program has to be loaded, the name and segment identification added to the list, and the creation of the new process has to be attempted again.

This failure to find the name in the list implies that the process which sent the message to the Program_manager has to be queued until the program is loaded. The Program_manager itself cannot perform the loading of the program since this would require it to send messages to the file system servers, which themselves must create
processes and send to the `Program_manager`. Here the first worker process in the system is required.

The `Program_loader` is a worker for the `Program_manager`. Its task is to send a message to the `Program_manager`, requesting that it be given the path name of a program to load. When given such a name, it deals with the file system to have the program loaded, has the kernel convert the shared segment into a program segment, and sends the segment identification with the name back to the `Program_manager`. Should the program not be loadable for some reason, this information rather than the segment identification can be returned. Diagrammatically the program used by the `Program_loader` can be seen in Figure 4.34. As with most worker processes it tends toward simplicity.

```
Request = READY_TO_LOAD;
Send( PROCESS_MANAGER, Path, LENGTH, MODIFY );
REPEAT
  if file=Access(Path)
    size = Size_of(file);
    id = New_Share( size, WRITE );
    if id == 0
      Reply = NO_MEMORY;
    else
      Load_Program(file, id );
      id = Make_Shared_a_Program( id );
      Reply = SUCCESS;
      Segment = id;
    else
      Reply = DOES_NOT_EXIST;
      Request = LOAD_STATUS;
      Send( PROCESS_MANAGER, Path, LENGTH, MODIFY );
  FOREVER
```

Figure 4.34 Program_Loader

The contents of the subsidiary functions called in the program should be deducible from their names. Some of the details have been left out for brevity. The discussion of the `Program_manager` can now resume.

The `Program_manager` is a simple administrator style process. When the `Program_loader` sends a message announcing the successful loading of a program, the `Program_manager` can add the new name and segment identification to its list. It then can run through its list of queued requestors, attempting to create any processes which required
the newly loaded program. A failed load runs through the queue of requestors, replying with a failure note to all those waiting on the loading of the program in question. If the queue of requestors is not empty, the Program_manager has the Program_loader load another program. Should the queue be empty, the Program_loader is not replied to but left until the next time it is needed.

Only a request from client processes to have a new process created has been discussed to this point. The Program_manager has to handle other requests. One very important request, which is not immediately obvious is the FORGET IT request. Should a program be recompiled, the copy which may be in memory is no longer the most recent. It would be frustrating to a programmer to discover a feature in a program and fix it, but have the system continue to use the old program just because it was loaded in memory. The list maintained is decreased when a program is found to no longer be in memory, and also when it is no longer valid.

The two requests already covered are all that is needed for complete management of programs. It is all very well to state, "added to the list of ..." and, "simply queued until ..." when discussing the Program_manager but the use of address space has to be seriously considered. The Program_manager exists forever. If it occupies too much memory, that memory is removed from use by all other processes.

The Program_manager has two distinct lists which are, potentially, unbounded. The list of loaded programs tends to increase in size but the list of queued processes tends to be empty. Interest should be directed to the list of loaded programs first.

Consider the situation where there are 1,000 programs stored in files with long path names. If some process makes a pass over these programs creating a process on each, the Program_manager will have 1,000 program names and segment identifiers within its list. This can account for a reasonably large amount of memory and quite likely will contain entries which are no longer valid. Carried to its extreme, most of the list can be invalid due to the fact that, because the list is so long, there is no other memory in the machine to hold the loaded programs. The growth of this list must be controlled in some manner.

The list of loaded programs will grow when a new program is loaded. This new program will be loaded when it has been discovered that it is not already loaded. This appears to be the perfect point at which the Program_manager can make a swift pass over the list, removing any entry which has an invalid segment identifier. The list will be
trimmed back to a valid state every time a new program must be loaded. Having only two entries using 500 memory locations in a data segment which is 128K in extent is of no use to anyone. The extent of the list as well as its length should be controlled.

The simplest method of keeping the extent of the list to a minimum is to move all entries which occur at higher addresses than the entry just removed. While easy, this can lead to a wastage of time. If movement is delayed until all invalid entries are identified, fewer entries will be moved. The algorithm for compacting the data area is clear.

The list is kept sorted by increasing address. After all invalid entries are removed the list is again traversed, moving each entry to the lowest address available. This movement is only done if the size of the areas unused is going to make the data segment sufficiently less than half the current size. After all entries are moved the size of the data segment is reduced, providing free memory for other uses within the machine. This reduction of segment size is quite inexpensive with a buddy system of memory allocation. Half of the segment becomes a free segment. As doubling the segment size is potentially more expensive, this is not carried to the extreme. It would be foolish to allow a situation where the segment size toggled constantly between 2048 and 4096 because the used amount toggled between 2000 and 2100. This automatic release of unused space can be restricted to larger segment sizes, and appropriate values used to match the word "sufficiently" which appears above.

The handling of the loaded program list is straight forward. The list of queued creation requests is not. This list can conceivably contain almost every process in the machine. The only processes which are not candidates are those which are required to be active in order to have a program loaded. If the Program_manager is prepared to handle such a long list, all is fine, but can account for a large amount of memory. While the worst case limit for the length of the list is large, the expected length is much less. Under normal operating conditions it will probably never be more than a few entries long. Such a situation comes about by the algorithms used in normal programs.

In general, when a large number of processes are to be used to accomplish a given task, the most convenient way to organize their creation is by the means of one process which creates all the others. This provides the coordination which is usually needed. In the situation where a large number of new programs have to be loaded, they tend to be requested by one process and, while that process will appear in the list of queued processes many times, it will only use one entry. For the situation where very old programs are requested and these programs have been discarded, again the probability that a large number
of these will be requested at the same time is slight. The implications of this is that the maximum size of the list should be quite small. The overhead to deal with a dynamic list appears to be too great to justify its use, but the number of entries in the list is something which can only be determined when the complete system and machine is in service.

The exact number can be made changeable without being what would be considered dynamic. Any system should be built with the facilities necessary for measurement and performance evaluation. The Program_manager will be collecting statistics on its performance. It becomes a simple task to provide evaluation programs which can collect these pieces of information and make deductions as to required changes.

This takes the discussion to the next request that the Program_manager, and indeed all servers, should handle. The Program_manager will accept a GIVE_INFO request, respond with the information it has collected since the last time the request was received, and reset its statistics variables.

This request implies yet another. The Program_manager should accept a SET_PARAMETERS request which can change such things as the maximum size of the queued request list. If the list is stored in memory after the loaded program list, decreasing or increasing the size is simple. The Program_manager can adjust with changes over time. There is yet the issue of what to do if the list of queued requests is full when another arrives.

In general fixed length queues can be supported with receptionist processes which can be used to easily control the situation where the queue length can be exceeded. That is not reasonable here as it would add extra complications to the situation. For example, if a process needs to be created to have a program loaded, there must be some means of allowing the file system to avoid the congestion of the receptionist. Since the process whose creation is requested by the file system must never need to be loaded from the file system, the request will never cause an addition to the queue and so should be allowed through. A simpler solution is possible.

Since the list of queued requests should "never" fill, the way to deal with the situation where it does fill, is to provide a delaying response to the process which could not be queued. The process which sent the message did so by calling a library function which performs the sending of the message to the Program_manager. This library function can
be written to accept a **TRY_AGAIN** reply. Given such a reply, it can delay for a short period of time, and then repeat its message.

The remaining problem is how to get the **Program_manager** to reduce its memory usage when it can but does not "think" it should. This implies that a final request it must accept is a **SHRINK** request, which will cause it to reduce its memory usage. All system processes should be prepared to accept this request.

![Diagram of Program Management Processes](image)

**Figure 4.35 Program Management Processes**

Figure 4.35 shows the two processes needed for program management. The **Clients** processes and the **File System** are shown as rather nebulus areas. More will be seen of the **File System** next, but for now it can be noted that the **File System** is part of the client set. The arrows in this figure, labeled A, B and C, represent the messages which are sent.

The **B** messages are **READY_TO_LOAD** and **LOAD_STATUS** as seen in figure 4.34. The **C** messages are those required to access a file, read its contents, and terminate access to the file. The **A** messages have already been covered. It is these messages which define the external appearance of the **Program_manager** to all other processes.

Provided the external appearance of the **Program_manager** is maintained the implementation can change. The **Program_loader** need not even exist. With time the external appearance of the **Program_manager** will most surely be extended with new requests being accepted.
Section 4.4 File System

With any file system, for any machine, there must exist some means of naming files. It is by the means of these names that most people, and programs, identify specific files. The set of facilities for naming files is a reasonable place to start the discussion of the file system. It can be assumed that the file naming scheme is at least a hierarchical one.

4.4.1 File Naming

At first glance it would appear that one logical file system, with any physical divisions hidden from the user, is a desirable goal. This is both valid and invalid. Most of the time there is no need to know what physical device holds a file, or whether it is on the local machine or stored on some other machine in a network. Conversely, knowing whether a file is on a permanent storage device or on a removable device is important if the removable device is going to be removed. Further, considering the possibility of moving a file from one named location in the file system to another, should both names fall within the same physical file system, the file may move by altering the position of the file within the system. Should the two names span disjoint physical file systems, the file must be copied, and the move can possibly fail due to space limitations. The requirement is for a single logically transparent file naming scheme, which can be easily interpreted as a collection of distinct file systems when desired.

The simplest means of providing these two conflicting requirements, is to use the first name of the full hierarchical name as the indicator of the distinct file system to be used. For convenience this full hierarchical name will be referred to in further discussions as a path name. It specifies the path to follow through the file system to reach the desired file. Here the first process structuring of the file system support can be seen.

There exists one process which accepts all requests dealing with files by path name. This process accepts requests from any process. Only this process need register as a server. It has, as workers, a set of processes which handle the name requests for individual physical file systems. This structure can be seen in figure 4.36.

The File_namer is an administrator style process. Its internal structure can be well defined. When it receives a request from a client process, it must save the important part of the message in a queue associated with the named file system. When a worker process
sends a message with the response to the task it was previously given, the appropriate client process is replied to and the data in the queue removed. When a worker arrives for the first time, a new record describing the existence of the worker has to be created.

![Diagram of File System Processes]

**Figure 4.36 Initial File System Processes**

The worker description record needs to hold a small amount of information. Since the **File_namer** has to transform a textual name into a worker identification there is the need for the textual name for that worker to be stored. There must be a pointer to the first queued request for that worker, in case there are queued requests. The process identifier of that worker needs to be stored as well, so that the **File_namer** can identify which process is the worker for the given name. The final piece of information needed is an indicator which can record whether or not the worker is waiting for work.

Given that the **File_namer** does not deal with client requests, there is no need for it to understand any of the fields in the message which is sent by a client process, other than to deduce the required worker. If the clients are required to place the first name of the full file name in the text section of the message, the **File_namer** need store only the rest of the message in its queued requests. The first name of the path name will have served its purpose in identifying the correct queue. Along with this message section, it needs to keep the identification of the client process, and a link to the next queued request.

The worker list, while increasing as new workers report for the first time, will tend to be a fixed size, nor is there any need to order the elements in the list. The queued requests
for each worker can vary in length quite wildly and need to be of a more dynamic nature than the worker list. Considering that the queues can become large at certain points, the File_namer should support a SHRINK request, just as the Program_manager does. Theses facts lead to a simple organization of the data stored by the File_namer.

The File_namer has four variables, Workers, Worker_count, Free_queue, and Queued. Worker_count exists for convenience. Three of these variables point into the dynamically allocated segment of the File_namer. The dynamic segment first contains the worker records and then the queued requests. The variable Workers points to the start of the dynamic segment. Queued points to the area used to store the queued requests. Free_queue points to the first queue entry which is not currently used to store a request. This list of free entries is kept sorted by address the address of the nodes themselves. Initially the dynamic segment is made big enough to hold one queued entry, Workers, Queued, and Free_queue are made to all point at the start of the dynamic segment, and Worker_count is set to zero. This initial memory layout can be seen in figure 4.37.

```
<table>
<thead>
<tr>
<th>Workers</th>
<th>Worker_count</th>
<th>Queued</th>
<th>Free_queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 4.37 Initial File_namer Data Structures

The first thing to consider is the means of distinguishing worker requests from client requests. In general, a message is identified by the value of the request it contains. Workers can be distinguished from clients by the requests they use. This was seen previously when program management was covered. Here the situation differs sufficiently, so a different means must be used. The File_namer has to pass back, to clients, the responses from the workers who perform the required tasks. If workers used the request field of the message to indicate that the message was a response to a client, the File_namer would have to set the request field of the response to the real response value, which would have to be stored in some other field of the message. This would introduce the need for some knowledge of the response in the program of the File_namer, and open a hole for subtle bugs.

Assume that some faulty program of some client program inadvertently gets a "bad" value into the request field. The File_namer would treat the request as a worker's response, and reply to some other client with some obscure message, and the "fake" worker
would be given the request of the next client on the queue as a response. When the real worker provides the response to the client which has already been replied to, the next client on the queue would get the response. This seems a fertile area for obscure, difficult to reproduce problems. The safe way out of this is to have the File_namer check that the process which sent the worker type request is a valid worker.

Since the checking of the validity of the worker's identification is needed, it is simpler to identify workers by the fact that the process making the request is a worker. The File_namer need have no knowledge of the contents of the response to be given to the client. The full message received from the worker can be passed to the client as the response.

This solution is possible because the workers which are known to the File_namer only send one request. It is a, "Here is the response you should give, as a reply to the client whose message you previously gave to me, and give me the next client's message please." Considering the difficulty in coming up with a short meaningful name for such a request, there is, fortunately, no need to. As each request is received, the identification of the requestor is scanned for in the Workers list. If it is found, the requestor is a worker which sent one of "those" requests. Failing the match it must be some other type of request from a non-worker.

All requests from non-workers are treated identically. The File_namer searches down its Workers list, looking for a match between the stored textual name, and the textual name in the message it has just received. Either a match is found, or it is not. If no match is found, the request is a special one destined for the File_namer itself.

The first such special request is a WORKER_REGISTERS request. A new worker record has to be created. The dynamic segment is increased to provide space for the new record, if necessary, and all pointers into the area pointed to by Queued are incremented by the appropriate amount. The area pointed to by Queued is then moved by the same amount, opening the area for the new worker record. This new record is initialized and then treated as if it always existed, with the new worker waiting for a client.

The second request to consider is the SHRINK request. Because the amount of memory used to hold the queued requests can have become excessive in the past, it is possible that the File_namer may be able to decrease its memory usage. It is worthwhile checking to see if there is a possibility to decrease the memory assigned before attempting to
compact the dynamic segment. This is easily done since the length of the Free_queue list, multiplied by the size of each entry will give the number of addressable units that could be freed. If this would not result in any freeing of memory there is no need to compact.

Given that compaction is useful, it is easy to do. Proceeding down each of the stored queues, if the address of the queue entry is greater than the address of the first entry in the Free_queue, the free entry is removed from its queue, the entry is copied to that free entry, and the space used for the copied entry is inserted into the Free_queue list. The reason for keeping the Free_queue list sorted by address is now obvious. When all saved queues have been processed, all free queue entries will be at the end of the dynamic data segment. The last free entry which will be retained can have its link field set to a null pointer, and the unused memory given back for use by other processes.

Another common request that the File_namer should handle is the GIVE_INFO request, for the same reason that the Program_manager handled it. Information about such things as the maximum length of any worker queue, which worker queue had the maximal length, number of times it managed to respond positively to a SHRINK request, etc. can be of use in analysing the performance of the system.

If the request matches none of the acceptable special requests, a NO_FILE_SYSTEM response is generated rather than simply terminating the requesting process. This is chosen as it is assumed that the only time such a situation would arise is if the request were a genuine client request, but the named files system did not exist.

If the request received was not from a worker, and the textual name in the request matches one of the worker records, the request has to be added to the queue of the worker record. Given a free queue entry, the free entry can be set to the message contents, and added to the tail of the queue. With no free entry, the size of the dynamic memory segment has to be increased. If the segment cannot grow a TRY_AGAIN reply is given to the client, just as in the situation with the Program_manager. This is a situation which will "never" happen, but has to be dealt with when it does.

Given the ability to queue the request, the worker record in question is marked as the active record. The common handling of the active record will take care of any situation where the worker was waiting. Only if the active record describes a waiting worker, and a queued client, will the common handling do anything. This will be true if the worker has just sent the current request, or the client was added to an empty queue. There can only be
one active record for each request received, and that record need not always result in any immediate activity.

Moving on from the **File_namer**, the next process of importance is the **Namer**. There is one **Namer** process for each file system. All **Namer** processes need not use the same program. In fact, it is advisable if all did not use the same program. When normal magnetic media, **WORM** disks and remote network file systems are considered, it is obvious that unique methods of handling these vastly different problems are advisable. All that is important is that, to the outside world, all **Namer** processes react in the same manner. To simplify the discussion only the **Namer** for a normal magnetic media file system will be covered. A network **Namer** is a front end process for a **Namer** on another machine and can be ignored for the time being. A **WORM Namer** is much like a magnetic media **Namer**, but uses data structures which reflect the nature of **WORM** disks.

There are three basic tasks that a **Namer** process has to handle. Two of these require one path name, while the other requires two. A client process can request access to a file, access to a directory, or the movement of a file from one named location to another. The two access requests will result, if successful, in the return of a process identifier. This process is to be communicated with for all operations on the file or directory. A path name is changed into the identifier of a process which will manage the file for the client. Moving a file from one named location to another will result in only a status indication.

Every path name specifies up to two files. Rather than making an artificial distinction between files and directories, every valid path name indicates both a directory file, and a data file. If a path name has other files under it, it will have a directory file associated with it. If there is data stored at that path name, it will have a data file associated with it. Which is being accessed is determined by what the access is for. When given a path name for access, the **Namer** treats the path name as consisting of two distinct parts. There is the last component of the path name, which specifies the exact data or directory file in question. All other components of the path name indicate directories which are to be traversed to eventually reach the directory in which the specified data or directory file will be found.

Processing the first part of the path name is relatively straight forward. For each directory which is to be searched, the directory file is accessed and scanned for the appropriate name. When the name is found, this indicates both a new directory file and data file. The procedure is repeated until the directory containing the specified name is encountered. Processing every path name request from the root of the file system, by
reading every directory in the first part of the path name, can consume a considerable amount of processor and I/O time. Many systems avoid this by associating a "current directory" with each process. The process in question can then specify path names as "relative" to the current directory, saving the costs of using a full path name. For example, if the current directory of a process was /a/b/c/d/e/f in such a system, accessing /a/b/c/d/e/f/g would require looking for g in the current directory. This is an appreciable saving. If half of the file accesses by this process are under the current directory, but the other half are to a sub-directory of the current directory, half will require searching both the current directory and the sub-directory. Should other path names be used which are not under the current directory, these may have to be processed as full path names. Current directories provide an efficient means of handling some of the access requests.

An alternative to the use of a current directory is to always use full path names, but to remember the results of previous searches. This allows further accesses under the same directory to be handled without searching in earlier directories, and allows more than one such saving path. What needs to be saved is the name searched for, an indicator of the parent it was found in, and the two files associated with that name. The first time that the path name /a/b/c/d/e/f/g is encountered the information of each element is stored in a record contain the fields seen in figure 4.38. Note that files are remembered by their file number. This is how the lower level file system, as will be seen, "names" its files. When /a/b/c/d/e/f/g/h/i is to be found, the directory associated with g has to be searched for h, then the directory h has to be searched for i. The path name /a/b/c/d/e/f/g/h/j will require searching the directory h for j. The path name /a/b/c/d/e/k would require searching the directory e for k. This is where the real saving of this method over a current directory style starts to appear.

![Figure 4.38 Namer Cache Entry](image)

The "caching" of directory entries provides an adaptive set of current nodes, and is more suited to a system composed of discrete communicating processes. To maintain a
"current directory" in such a system the file system would have to keep information about every process which exists. It would either require the file system to track the creation and destruction of processes, the kernel to maintain this information for the file system, or the creation of a process to require communication with the file system to assure that the new process had the appropriate current directory associated with it. By far the best solution to any problem is to avoid the problem. The chosen solution approximates the current node saving, avoids the problem, and the "cached" directory entries are available to all processes, not just the one which caused the entries to be saved.

Two major questions about this "caching" have to be answered, but can be answered only after a working system has been used for some period. These two questions deal with the number of entries to be saved internally, and which entries are to be saved. These two questions are interrelated to a certain extent. The first question deals with which entries to save. Is it reasonable to save the final file or directory entry? If a file is to be accessed multiple times, the answer is yes. If files tend to be accessed once, the answer should be no, since storing those entries implies removing others. Those others may turn out to be directories which should have been kept. The situation is somewhat analogous to the page replacement problem in a paged memory environment, or the ordering of symbolic names in a lexical scanner which uses hashing with chaining. It seems obvious that multiple accesses to a file are probable. Editing a file implies accessing it for reading, and after the editing is complete, accessing it for writing. The times between the two accesses can be great and during that time enough other accesses can be made that the cached entry can have been lost. What confuses the matter is that there is no distinction between files and directories. It can be assumed that a path name which does not name a node with substructure, will probably not be accessed again, soon. If there is substructure, that substructure may very well be accessed soon, and so the terminal name should be saved. Again this is complicated by the number of entries saved. If too few are saved, keeping entries which "possibly" will be searched may force useful entries out. The real nature of the problem cannot be seen unless one further point is cleared up.

It has been stated that each named entry in the file system may specify up to two files, a data file and a directory file. This statement no doubt raised in the reader's mind, a visualization which need not be exactly correct. A multi-way tree can be supported by either a directory file, or by a binary tree representation. It is obvious that a simple implementation of both will show that the use of a directory file is the better choice from a performance point of view. Searching for a specific file in a directory will, in general, require fewer disk

156
operations than a linear search of the right branch of a binary tree which can be scattered over the disk. Two basic problems of a binary tree representation can be summed up in the words "linear" and "scattered".

Given the ability, and opportunity, to rearrange the position in storage of the nodes used in a binary tree representation, the amount of scattering can be greatly reduced. The algorithm to do so is quite simple, and requires the ability to interchange two stored nodes, and necessitates the updating of two others, those which point to the stored nodes. The algorithm is repeated until all right branches of any node are sequentially stored. For a personal machine, the opportunity arises over many hours of each day. When the machine is not being used, such as overnight, there is more than enough time to complete the required operations. Done properly, this activity can continue while the machine is actively being used. Again, collected statistics on the number of out of sequence operations, and the total number of operations, can be used to decide when it is advantageous to adjust the physical storage. Much of the performance loss in using a binary tree representation over a directory file representation due to the scattering problem can be overcome. The other area of interest is the linearity of searching the right branch of the tree.

If the number of entries in the right branch is small, there is very little concern about the linearity of the search time. It is when the list gets long that it is worth considering. If the terminal entries in a previously looked up path name are stored in the cache, and the same path name is again given, the linearity argument is not relevant since no searching is needed. Accessing the same file will, in general, not be the case. There is locality of reference to be considered here. Files tend to be placed in the same directory because they have something in common with the other files already there. Tree structured file systems are popular because people do organize the things that they store. If one file is requested from a directory, another file from the same directory is possibly going to be requested soon. With a linear list of files, when the first file is requested there is no alternative but to search linearly for that entry. If the entries in that list are ordered by some function of the name, information about previous requested files can be used to "shorten" the list. If the file with the name Y is requested, but is not known, but the file with the name X has previously been requested, and Y should come after X by the definition of the ordering used, the search can commence with the entry after X, rather than at the first entry in the list. If the order chosen is the same as the most popular order that files are accessed, the linear search is much reduced. A reasonable first ordering to consider is by dictionary order.
When the list of files under a node are presented to a human there is some order to that list. The order applied tends to be alphabetic since the location of a specific name within the list tends to be stable, and a person can easily use this ordering to locate any name in the list. For example, if a directory holds the chapters of a book, the names of the chapters will tend to be numeric. Production of the whole book will reference the files in the perceived order. If the perceived order is the stored order, the files will be referenced in the stored order.

It should be noted here that one directory which is accessed in a rather random order, is the directory which stores the executable commands. In this system the number of accesses to that directory are lessened greatly by leaving the loaded command in memory even when not in use, provided the memory space is not needed. It reduces the number of multiple accesses to each file, but not the number of unique accesses. The directory of commands tends to be one of the larger directories. If all names of the specified path name are stored in the cache, the ordering can be used to reduce the amount of linear searching needed to find a specific file. Experiments with the Thoth system which used an unordered binary tree solution showed a difference in search times of the command directory of over one second between accessing the first and last commands on the list. This was one of the determining factors in the subsequent Port system going to a directory file system.

For a system with a directory a non-linear searching solution is possible. Binary searching of an ordered list is trivial. The creation or deletion of a file may require that a goodly percentage of the directory file be modified. This may require the modification of multiple disk blocks. This leads directly to a need to consider the robustness of each solution under conditions of failure.

If it is assured that the order of requested modification operations to the disk is the actual order of modification, a proper ordering of operations using a binary tree representation will provide robustness without excessive complications. Since each entry in the binary tree can be changed in an atomic manner the proper ordering can provide robustness even under situations where the machine may be halted at any point in time. If transaction processing solutions are not used, the interrupted creation of a file will result in the loss of one free storage entry, as will the deletion of a file. Using a transaction processing solution, two extra disk operations are required to provide complete robustness. When the movement of a file from one named location to another is considered, two extra disk operations again provide complete robustness. These two operations are used to
bracket the modification of two entries in the case of creation or deletion, and four entries in
the case of movement. One disk operation is sufficient to record the intended changes, and
one for the completion of the changes. Due to the fact that the amount of storage changed in
any operation will consist, at the most of four entries, these four entries, and the associated
information needed to support the transaction, will fit into an atomic disk operation. In
implementation there is no need to perform both of the extra operations. The one which
precedes the modifications records both the intended changes, and the fact that the changes
are to be made. The one which follows the modifications records that the changes have been
made. Until any of the nodes in question are again modified, the information stored in the
transaction record is correct. Not recording the completion of the transaction will require the
completion of the transaction when the machine is restarted after having been halted.
Repeated application of the transaction by its very nature is safe. The only time the recorded
transaction must be marked as completed is before any of the nodes in question are again
modified. Since this transaction record is changed to record the next modification, before
the next modification is done, there is no need to record when the modification has been
completed. This results in one extra disk operation for each modification, to guarantee
complete robustness. Providing robustness in a directory system which attempts to order
the entries in the directory is a much more complicated and expensive procedure.

It is unfortunate that robustness must be assured by requiring even one extra disk
operation. Recording the transaction will in all probability require head movement on a fixed
head disk. This can easily be remedied by the provision in the machine of a section of non-
volatile memory which can be used to store the transaction record. The reason for storing
the transaction record on the disk is to achieve this non-volatility. Given the existence of a
"non-volatile memory" proprietor the simplicity of robustness provision in a binary tree
system argues for its acceptance.

This robustness in a binary tree system is gained at the result of some performance
loss over that of a directory system, but accepting errors for the sake of speed seems to be
an invalid position. It is possible to make a directory file system robust. It is just more
complex since the pieces of information which are being safeguarded are found at some
offsets into some files, and are of arbitrary length. To assure that recording the changes in a
directory system can be done in an atomic action, ordering entries cannot be done.
From the above arguments, to return to the original point, it seems that all names on the full path name of a file should be recorded in the cache if a binary tree representation is used. If a directory file representation is used, this is probably not worthwhile.

The initial method of recording the structure of the file system is by way of the binary tree method. The solution is simple and robust. All path name elements will be saved in the cache. Should performance indicate that directory files are a better solution, a conversion is easily made. Once the intricacies of robustness are solved, all files can be saved on an archival device, the file system initialized in the new format, and all files restored.

It will be useful, when the system is in existence, to again have statistics gathering code which can be interrogated. The entries of the cache are in an ordered list. When a new entry is to be added, the one at the end of the list is lost. If statistics are kept for the number of times each entry position in the list was reused, and the number of times a new entry was needed, it should be possible to arrive at a reasonable number of entries. If the entry list is too long, the percentage of reuse near the tail of the list will be low.

Identifying the file or directory to be accessed has been adequately covered. What has not been covered is the checking of permission to perform the requested operation. Now is a convenient point to do so.

At first glance there seems to be little need for permission checking for a machine sitting on one person's desk. One should be able to do what one wishes with one's files. The real need for permissions is the prevention of inadvertent deletion of files. For file servers on a network there is a need to provide at least as much in the way of permissions as is found in many multiple user systems. Given that any machine in a network should be capable of allowing controlled access to a portion of its files, this implies that all machines should provide permissions checking. This brings up a number of issues.

The first aspect to consider in permissions checking is the time of checking. Given that the full path name is used for each access request, and that permissions should apply to the smallest area possible, it would appear that the last permissions encountered in the access path should be those which are applied, and they should be applied when the full path name has been processed. This allows control of access to sub-trees of the file system without regard to access to larger sub-trees.
When considering how permissions are to apply to persons, it is common to consider the person as an individual, as well as a member of a group. Many people are members of more than one group. The real question is, "Does this permission apply to this person in any conceivable guise?" Rather than making an artificial distinction between groups and individuals, each person can be considered to belong to a group which includes one person, as well as any other groups to which that person belongs. Each person has a list of groups to which that person belongs. Belonging to multiple groups has its problems. The major problem is deciding, on a given access permissions check, what group is important. If it is possible to function acceptably without resorting to groups, it is advantageous to do so. That eliminates one set of structures and simplifies the final solution.

There are two distinct aspects of permissions handling. One is checking, and the other is modification. Normal operations dictate that checking be as efficient as possible. Here the types of permission become interesting. There are general permissions and specific permissions. If only additive permissions exist, much checking can be avoided. Consider a sub-tree which has general permissions which allow read access. Should the requested mode of access be read, there is no need to see if specific permissions apply. Should subtractive permissions be supported, the full list of permissions may have to be processed to see if the person responsible for the request has been denied read access. The situation becomes more complex if the person belongs to more than one group. The question arises of what to do if one of the groups is denied read access, while another is not. The existence of subtractive permissions implies that full permissions lists have to be searched on every access, to assure that the individual is not named. In general the existence of subtractive permissions implies excessive overhead for little gain, and can be ignored.

Usually the general permissions are sufficient for most checks. Consider a sub-tree which can be read by anyone, but is restricted in the list of people who can modify it. The vast majority of access attempts will be valid. Even for those with modify permissions a large number of accesses will be for reading. If the cached entries include general access permissions, when access is checked, the information necessary may already be available, and the checking is swift.

For a personal machine there is one individual using the machine at any one point in time. This implies that should the general permissions not allow the access requested, but the specific permissions do, storing the specific permissions, rather than the general permissions, in the cached entry will permit rapid checking of future accesses. For server
machines on a network, the replacement of general with specific permissions is not reasonable, since most accesses can be satisfied with general permissions, even when one access which required specific permissions has been made. Both the general and the last specific permissions used can be stored with the cached entry.

Storing permissions at every node in the tree can be excessive. There is generally no need for such fine grained access restrictions. The scheme used in the Port system, where specific file types indicate the intended usage of a file, and the specific type Lock, implies that the file indicated contains permissions, is sufficient. The permissions scheme is simple. As each element in the path name is processed the type of the node is saved with the entry in the cache. When the full path name has been processed, the last Lock type file is remembered. If the general permissions are not recorded, the lock file is accessed and the general permissions recorded. If the general permissions are not sufficient to allow access, the file is scanned in an attempt to find the applicable specific permissions, and store them in the cached entry. If the stored general or specific permissions in the cache entry permit the attempted access, access is allowed, otherwise access is denied. Future access to all files under the specific Lock node can then be checked with no recourse to the permissions file, as long as the cached entry is retained. Should very fine grained permissions checking be desired, more Lock files can be included in the structure.

For a personal machine, determining the individual in question is simple. For a server on a network this is can be more difficult. In general, there has to exist a representation of some number of distinct individuals at any one point in time. This can easily be accomplished with minor support by the Kernel and the Program_manager.

Every process has a user number. System processes have a user number of zero. When a process is created a user number is supplied. This is where the support of the Program_manager comes in. It must pass this information to the Kernel. If the process responsible for the creation has a user number of zero, the supplied user number is used to set the user number for the new process, otherwise the new process is given the same number as the process responsible for the creation. All file accesses from remote processes have to pass through local processes. For each remote machine there can be one file access process which will have been created with the appropriate user number. The user number of the process requesting the access can be used to identify the specific permissions in question.
Access checking has been covered and now attention can turn to modification. The modifications of interest are not those of giving or revoking permissions on specific sub-trees, but the introduction or removal of specific individuals. If permissions were specific to each file, giving or revoking permissions would be expensive if groups were not supported. To provide access to the source of a compiler for a new person assigned to maintenance of the compiler, that person would have to be added to the permissions list for each file of source and object of the compiler. Groups would simplify that task to adding the person to the appropriate group. A person leaving the compiler group would entail equivalent work to that of addition. It is these file-specific permissions schemes which make groups attractive in common systems. For the scheme proposed here, the individual need only be added or removed from a small number of Lock nodes. If the concept of groups is not implemented, a finer control of access is not only possible but encouraged. Consider the compiler example again. Were groups supported, it is probable that the source and executable files would all be modifiable by persons in the compiler group. The novice to the compiler group would inherit the full privileges of all compiler group persons, including the right to update the executable versions of the compiler. This could be avoided by forming multiple compiler groups and adding the new person to these groups as the situation dictates. The person would start in the "read only" compiler group, and progress up to the "fully responsible" group. There would be separate sets of groups for each distinct section of the compiler. For example, a person could be a "guru" for the code generator but a "reader" of the lexical scanner. Tightly controlled these groups rapidly reduce to covering one small sub-tree each. Considering the overhead involved when an access is made of determining which of the groups an individual can be considered to be in, and the overhead of storage of the lists of groups an individual belongs to, it seems to imply that groups are not needed. As a person's responsibilities vary, so do the specific permissions for that person within the appropriate Lock files. Rather than indirectly giving a person permissions to various sets of files by placing them in various groups, they can directly be given permissions by naming them in the appropriate Lock files.

Whether groups exist or not, removing a person from a system requires passing over the full file system and removing them from any permission lists where they are individually present. Groups bring no benefits, but imply costs, and need not be considered.
Accessing named files has been adequately covered. Permissions checking is simple yet thorough. The use of cached entries provide rapid path name searching, as well as potentially inexpensive permissions checking.

It is worth turning, at this point, to make a few comments about Namer processes which have little to do with file systems. The discussion has focused on the File_namer as a front end to possibly multiple file systems, some of which can be remote. It may function as the front end for other services.

Consider a machine which has access to multiple printers. There is one daisy-wheel printer, two inexpensive dot-matrix printers, and three laser printers of high quality. A user will want to specify a printer on which data is to appear. Some times this specification is exact. At other times there is little need for identifying the exact printer in question. This set of printers can be made available by "pretending" to be just a sub-tree of a file system. For this example the list of all files in the sub-tree might appear as:

```
printer
printer/daisy
printer/matrix
printer/matrix/programming_1
printer/matrix/programming_2
printer/laser
printer/laser/management
printer/laser/programmers
printer/laser/customer_support
```

If printer/matrix is accessed, the printer controlling process can decide which of the two is best suited to printing the file. These decisions can be based on expected service times, or geographical considerations. Alternately, the user can specify exactly the printer in question. Integration into the perceived global file system is achieved in this way, without any interaction with the file system. There are no "funny" files in the file system to support the printers. The set of printers can be scattered over multiple machines in a network. When a named "file" is accessed what is returned is the identification of a process to communicate with. It need not be a process which controls a file.

As a further example, the "path name" Office/diary may give access to the records describing phone messages, and the availability of members of the staff in an office. In this case, what the process identified provides is controlled access to entries in a database.
4.4.2 Space Management

Once an access has passed through the name service it reaches the space level. The space level is not concerned with such aspects as convenient naming of files or permissions checking. Its sole purpose is to manage the space used to store files.

As before, the basic style is that of the Port system. Each file is defined by a single file descriptor. A descriptor contains a set of pointer-extent pairs which define which segments of the mass storage media are used to hold the contents of the file. If the file is small enough the area of the descriptor used for the pointer-extents is used to store the contents of the file instead. The file holding the file descriptors is defined by the first file descriptor.

When an accessed file has to be grown it is doubled in size, under a few constraints. If free space exists adjacent to the end of the last extent, the last extent is extended by either the previous size of the file, or the size of the free area, whichever is less. If there is no free space adjacent to the end of the last extent, a new pointer-extent pair is used, and the extent is the minimum of the doubled size of the file, or the largest free extent. When the modification access of a file is terminated any excess space, allocated but not used by the file, is returned to the free list.

If a file cannot grow because all pointer-extent pairs are used, compaction is required. One form of compaction is attempted before all pointer-extent pairs are used. The pointer-extents are considered by pairs. The minimum pair of adjacent extents are found, and a section of free disk large enough is searched for. If one is found, the pair of pointer-extents is merged into a new, larger pointer-extent and the data copied to the new section of disk. This will free one pointer-extent pair. With twenty five pointer-extents available, this would be attempted when the nineteenth was used. Doing so provides a buffer of five for growth. Failure to compact the file so that less than twenty extents are used is a reasonable signal that the file system needs general compaction applied to it.

General compaction can be applied during normal operations. Cunning schemes can be tried, but a simple method works well. The requirement is one free file descriptor and the knowledge of which block is the first free block on the disk. The free descriptor is used to hold the definition of a new copy of a file. The knowledge of which block is the first free is
used to avoid useless operations. The ultimate goal is for all files to use one pointer-extent, and for all free blocks to be clustered together.

Consider each file descriptor in turn. If the descriptor is not in use, pass to the next. If the descriptor describes a file which uses one pointer-extent, and that extent occurs before the first free block, pass to the next descriptor. Having failed the first two tests, the file should be moved, either to reduce the number of pointer-extents, or to move it closer to the beginning of the disk. Copy the contents of the file into the "new" file. This is first done by attempting to set the size of the new file to the size of the old file. The space management will attempt to find the requested space in as few pointer-extents as possible, and as close to the start of the disk as possible. If the result is more pointer-extents, or the file is further from the start of the disk, the blocks for the new file are freed, and the next descriptor is tried. If the new file is "better" the contents of the old and new descriptors are swapped, and the space originally used by the file is freed.

The first pass over the file system will tend to introduce a large number of small holes, as files with many small extents are changed into files with one larger extent. The second pass will move many small files into these holes, from locations at larger disk addresses. Further passes can be used to approach more closely the ultimate goal. Compaction can proceed during normal operations by considering just those files which are not currently being accessed. Compaction needs no extra support from the file system than is required for normal operations. Figure 4.39 shows the algorithm a program would follow to compact the file system.

```
while file = Next_file()
    if Exclusive_access(file)
        if Set_file_size(TEMP,Get_file_size(file))
            if Better(TEMP,file)
                Copy( file, TEMP);
                Swap(file, TEMP);
                Free(TEMP);
```

Figure 4.39 File Compaction Algorithm

This general compaction algorithm will only fail if there is a file which is bigger than the amount of remaining free space. In such an abnormal case the best solution is to perform
a full back-up of all files, clear the file system, and restore all files. This should result in a clean, compact file system.

Returning to the implementation of the space level, one aspect to address is the limit on the number of open files. In general, most systems place a limit on the number of files which can be accessed at any one point in time. Some systems have a "per process" limit, while others have an overall limit. Of the two choices, the "per process" model is preferable because the programmer can be responsible for not exceeding the limit. An overall limit implies that a program can access a total of X files some times, while the limit will be reached with X-N at other times. Such a variable limit makes it much harder for a programmer to construct a fully functional program. A "per process" model does tend to reduce the potential number of files which can be accessed by any one process, since there is truly an overall limit that is just evenly divided. If a choice has to be made, it would be best to choose neither.

There is no predefined limit to the total number of files which may be accessed at any point in time. In principle, every file is identified by a "key". The "key" is based on three components. One is the file number. This is included so that the space level can easily identify which file is being requested. A second component is the "boot" number. This is the number of times that the machine has been started. The third component is the generation number, the number of times that the contents of the file have been exposed for modification. These three values are passed through an invertable function, to produce this file "key". When a process attempts to make use of access to the file, this "key" passes through the inverse function to reproduce the three components. The boot number is a generally known quantity. The file number identifies which file is to be used. The generation number is stored in the file descriptor. If the key reproduces the correct boot number, and the generation number in the specified file descriptor matches, the use is acceptable.

A file can be accessed in, essentially, three different modes. There is the access which is made to read the contents of a file. A second form of access allows the file to be appended to. The third form of access allows the contents of the file to be changed. When a file is accessed in a mode which supports change, the process identifier of the requesting process is recorded in the file descriptor, and the time of access is stored. This time forms the generation number of the file descriptor. When this modification access is terminated, the time of termination is stored as the generation number, and the area reserved for the
modification process identifier is set to an invalid process identifier. Any attempt to access a file which has a valid modification process identifier is refused.

There is no need to store anything in memory about which files have been accessed by which processes. A translated key either agrees with the information available, or it does not. Agreement allows use of the file while disagreement denies use. Exclusive use is possible by accessing the file for modification. Should the modifying process not terminate access before itself terminating there is no problem, since any attempted access will check the process identifier, and discover that it is no longer valid, and so the access can be allowed. This is where the boot number comes in. It is possible that a system process will always access a specific file for modification. It never terminates access. When the machine is restarted, the process will usually get the same process number since the starting sequence is probably the same. A special case could be made for a process which attempts to have concurrent modification access to a file more than once, solving one problem but forming the basis for others. Should the order of process creation be slightly altered, the old identifier of the process may be a valid identifier for another process, making such special case handling useless. If the boot number is the time that the space level proprietor process was created, and the modification time stored as the generation number of the file is less than the boot number, the process identification stored in the file descriptor is, by its very nature, invalid.

Any file descriptors, or parts thereof, which the space level maintains in memory are only kept for performance reasons. For example, it could "cache" the last twenty file descriptors which were used. Further use of these files would not require any disk accesses to obtain information. Should more than twenty files be accessed at a time, some of the uses of these files would require disk accesses to recover the file descriptors which were not stored in memory. The limit on the number of accessed files is a performance limit. Above a certain number of accesses, there will be a performance penalty.

An added benefit of the chosen scheme is that one process can access a file, and pass the key to a set of processes. Knowledge of which named files contain which data can be centralized, while allowing the distribution of use. For example, a single process can access data base files for modification, reserving for itself the right to access the files, and pass the keys to any processes it wishes to give read use. No other process can access the files since the modification access is still valid.
The space level is supported by a group of three processes. One is the space level manager itself. The other two are worker processes which assist in free space management. One keeps a description of which blocks of disk are currently not allocated to any files. The second keeps track of which file descriptors are not currently in use. These two processes can share the same generic program. They logically manage a bitmap.

When a new file is requested, the space level requests that the worker for free file descriptors find one free bit. When a file has to be grown, the worker for the free disk blocks is requested to find the appropriate number of bits.

The space level process has three queues onto which a client requestor onto may be placed when it arrives. Two of these are queues waiting to be serviced by the appropriate bitmap workers. The third is for requests which need not involve free space. This third queue has a maximum length of one. A client request will either be completely served, or will result in the client being placed on one of the two real queues.

In general, the bitmap workers should be capable of swiftly responding to work from the space level. There are two situations where this may not be possible, and they should be looked at.

Representing the free space as a bitmap is a viable solution. Keeping the same information as pointer-extent pairs is reasonable. Depending on the amount of fragmentation of the storage space, one method will be better than the other in terms of speed, and in terms of space. If the storage space is slightly fragmented, a pointer-extent solution can be faster than a bitmap solution, and use less data space. A seriously fragmented storage space is more compactly represented by a bitmap solution. It is conceivable that the bitmap worker process will switch from one representation to another depending on the level of fragmentation. For example, consider the effects of the above mentioned compaction algorithm.

Compaction will normally take place when the number of extents in allocated files becomes excessive. The free space at that point need not be fragmented excessively. It is possible that all free space is in one contiguous piece. As each file with multiple pointer-extent pairs is compacted into one pointer-extent pair, the free space will tend to become severely fragmented. As compaction continues, the free space becomes less fragmented until, if carried to completion, there is one contiguous piece of free space. The bitmap
worker may well switch between a bitmap representation and a pointer-extent representation, and back again, some number of times during the compaction. Each switch in representation may well remove the bitmap worker from active duty for a short period of time.

There is one situation where the bitmap workers will be "out of service" for an extended period of time. This is at the time the system starts.

It is possible to assure that the bitmaps are represented on the disk in a reasonably valid form at all times. When a file changes there are two independent pieces of information which must be updated to correctly reflect the change in block allocation. If the file grows, the blocks have to be marked as not free in the bitmap, and the file descriptor for the file has to be updated to record the new blocks. If the machine is turned off, doing these two operations in the correct order will just result in the blocks in question being "lost". Done in the incorrect order these blocks can be marked as both allocated to the file, and as free. The correct order is the reasonable one. Shrinking a file is handled analogously.

Another possibility is to not record the free blocks on the disk at all. Any changes to the bitmaps are restricted to the memory bitmaps. This assures that there will never be a time when blocks are either "lost" or multiply allocated. When the machine starts, the bitmaps are rebuilt from the information in the file descriptors. Any block not recorded as allocated to a file must obviously be free. If the machine stops at some expected point, the memory versions of the bitmaps can be saved to disk, with an indication that the disk copies are valid. Starting the machine will result in these bitmaps being read from the disk, and given to the bitmap workers. The first change to the bitmaps requires one disk operation to record that the disk bitmaps are not valid. Validity information could be stored by the non-volatile data proprietor mentioned earlier and so there is no real need for a disk operation to record the validity or non-validity of the stored bitmaps.

If the bitmaps on the disk are not valid when the machine is started, they have to be rebuilt. This is supported by a transient worker for the space level, the free initializer. The space level starts by creating the free initializer which creates the bitmap workers. The free initializer then initializes their data structures, either by using the valid information from the two stored bitmap files, or by passing across all file descriptors, extracting the required information. For a file system with many files it may be a few minutes until the bitmaps are completely regenerated.
By having the space level work as a form of administrator, any requests which arrive
during this rebuilding time, which do not require allocation or deallocation of blocks or
files, can be completely served. This means that, should the machine restart after a power
failure, it can be used in a reasonable manner immediately. The user can start editing files,
and displaying information long before the bitmaps are complete. Those requests which
require information from the bitmaps will be held.

One of the nice features of a message passing system, which uses forms such as
administrators with workers, is that many requests can avoid the critical path. A
conventional space level implementation might bar all requests until its data structures were
correct.

The rest of the space level management is quite common and is not worthy of further
discussion.

**Section 4.5 The Others**

There will be various other small processes which perform useful tasks. For example,
the non-volatile proprietor has been mentioned. There will probably exist a process to
"mother" the information which tailors a machine to an individual's custom definitions. This
is the reasonable place to have the definitions of such things as the expansion strings for
function keys.

Other areas of the operating system could be covered in detail, but that would tend to
be excessive, and in general inaccurate. An operating system is not designed, implemented
and used. After initial design, implementation and use, new information will become
available which can radically alter the system. Mention was made previously to the removal
of swapping from the Port system after it was operational and true information was
available. Being an operating system designed at the same time as the hardware implies that
as both evolve, the interaction will drive evolution in both directions. The needs of the
operating system will push hardware changes while the capabilities of the hardware will
give direction to the operating system.

It is hoped that the flavour of the operating system has been adequately covered.
Some details have been given for typical sets of processes within the operating system.
Completing the operating system is largely a task of extrapolation from what has been
given. Now it is time to again turn to the hardware.
Summary

This chapter has discussed aspects of the operating system in more detail. A scheme for classifying processes by their prototypical structure has been detailed.

The primitives provided by the kernel, and the structures it needs to support those primitives has been covered. The exact algorithm for the assignment of processes to processors in a multi-processor machine has been detailed. The algorithm effectively handles even the situations where differing processors may just be variations of a standard.

The processes which handle the loading of programs from mass storage, and the mapping of symbolic names to loaded program identifiers has been covered. Some of the general requests that many server processes should handle were discussed and illustrated with these processes.

The processes which manage symbolic names of files were covered, and the ease of extending such a scheme to symbolic names of objects in general was detailed. Various methods of representing multi-way trees were covered and an argument for the adoption of one was made. With it was a detailed method of assuring consistency during updates.

An example of a potential process for file space management was covered. A method of maintaining access rights without imposing a limit on the number of concurrently accessed files was given. With it was an algorithm to allow compaction of the free space if needed, while normal operations still continued.
Chapter 5
The Hardware

The hardware of any machine defines an area in which it performs best. More than the processor is important. The rate at which some workstations execute instructions is as great or greater than the rate at which some mainframes execute instructions. Mainframes sell because the other components clustered with the processor make it better suited to specific tasks than the workstation.

Closely associated with the processor are the other components on the same board which both enhance the basic processor, and define its view of the rest of the machine. Other boards contain the components which support mass storage, communications and other such device style components. Foremost among these in a workstation is the device which supports interaction with the user of the machine. All of these boards are connected together by some means so that information can flow between them.

Describing all components in detail would take an inordinate amount of space. Attention will focus on the components of the processor board, the memory board, the user interface board, and the bus needed to connect them.

Section 5.1 The Bus

The common route between all components of the machine is the bus. Should there be only one component which actively uses the bus, the bus can be a bundle of wires. By using synchronous principles the signals need be held on the bus for a specified time and correct use is provided. When more than one element attached to the bus is allowed to make use of it, some means of arbitration is needed to assure that there is only one user at any one time. If some means of detecting addresses that are not meaningful is required, there may have to be a control section. The control section could also be used to provide support for speed differences in memory by providing asynchronous operations. Any element attached to the bus, which can actively use the bus, can be termed a bus master. This includes the normal processors and the intelligent devices which on occasion need to control the bus. Bus masters come in two varieties. There are the constant users and the occasional users. Bus contention issues limit the number of constant users but the number of occasional users have no inherent limit.
The machine described here is a multiple bus master machine. There is a definite need to provide some means of arbitration. When a bus master wishes to use the bus, it must first assure that no other bus master is doing so. If the bus is free, it has to arrange that it is the bus master which is allowed to use the bus. It must indicate that it is using the bus, so that other bus masters can be aware of the fact that they are not able to do so. At this point the use of the bus has gone through the arbitration process and the bus master in question can use the bus as appropriate. When it has finished with the bus, the second aspect of arbitration has to come into play. The bus master has to indicate that the bus is now free for use. There are four parts to the arbitration procedure. Indicating its usage and the final relinquishing of the bus are conceptually simple. Since it has control of the bus this is done in a "single bus user" environment. The sensing that the bus is free can be associated with this aspect in a simple way. For example, assume that one wire is used for this purpose. By default the line conveys a low signal. If a bus master is using the bus, it holds the wire in a high state. When the bus master is finished with the bus, it stops holding the signal high, and it drops to a low level. If this wire has a high signal level, some bus master is using the bus, and all other bus masters wait until the signal goes to a low level before trying to use the bus. The true arbitration problem comes when a bus master attempts to indicate that it has control of the bus.

Many bus protocols exist to deal with the problem of bus arbitration. In general, most of them require that a bus master bid for use of the bus, and some external hardware arbitrates, and decides which is to be given the use of the bus. A means of providing this is to assign a unique signal line to each bus master for use as a bid line, and another line to each bus master as a grant line. Any bus master which wishes to use the bus raises its bid line and waits for its grant line to rise. When this happens, it raises the in-use line and lowers its bid line. When finished, it lowers the in-use line. The bus arbitration logic will guarantee that at most one grant line will be high at any point in time. If the in-use line is low, and there is no grant line raised, it will raise the grant line of one of the bus masters which is bidding for the bus. This situation can be seen in figure 5.1. At some time \( t_1 \), a bus master will assert its bid line. At time \( t_2 \), the bus arbitration logic will assert the grant line to that bus master. At time \( t_3 \), the bus master having detected that it has been granted the bus, will assert the in-use line. After allowing for propagation delays, at time \( t_4 \), the bus master will revoke its bid line. This allows the arbitration logic to, at time \( t_5 \), revoke the grant line to that bus master. When the bus master is finished with the bus it revokes the in-use line at time \( t_6 \), allowing the arbitration logic to choose the next bus master to be
granted use of the bus. At time t7, the bus arbitration logic is able to assert the grant line of the next bus master to be given the use of the bus. The time between t1 and t3 plus the time between t6 and t7 is the total overhead involved in the arbitration. The time between t3 and t6 can be overlapped with use of the bus, and is not important. Also, should any other bus master bid for the bus while it is in use, the arbitration logic can predetermine which bus master is to get the bus next, and so the time between t1 and t2 for the next bus use cycle may be absorbed into the current one.

![Diagram](image)

Figure 5.1 Simple Bus Arbitration

This scheme is appealing for a multiple bus master configuration, since the arbitration time penalty is essentially independent of the number of bus masters. Increasing the number of bus masters would increase the propagation delay due to a longer bus, but that is all. With a large number of bus masters, the arbitration time penalty is likely to be the sum of the propagation delays of the falling of the in-use signal and the raising of the grant signal. What is also important is the number of lines used. For N bus masters $2N+1$ lines are dedicated to bus arbitration. By the use of pulses rather than signal levels, and introducing extra complexity, this can be reduced to $N+1$ by using a single line for the bid and grant signals. The number of bus masters on the bus is limited by the number of lines dedicated to arbitration. Because a bus master may be an occasional user of the bus this is not desirable.

The number of lines on the bus can be reduced by grouping the bus masters together, and using multiple layers of arbitration. For example, if the bus masters are grouped into sets of eight, within each set arbitration will be performed, and between each set arbitration
will again be performed. For a bus master to be granted the bus the group it is in must be
first granted the bus, then its local arbitrator can grant it the use of the bus. By introducing
more complexity and delay a larger number of bus masters can be supported. The number
of lines is reduced, but still determines the maximum number of bus masters. In the
example given there are two sets of arbitration lines in the bus. One is the global allocation
set. The other is the local arbitration set. By breaking the lines of the local set at the physical
boundary of the local set, one physical line can serve multiple local sets. The centralizing of
arbitration requires either a large number of dedicated arbitration lines, or a longer
arbitration delay, with grants and bids traveling up and down chains of command. Neither
is appealing.

If the arbitration of the bus can be distributed, the number of physical arbitration lines
can be greatly reduced. If this can be done with a reasonably small arbitration delay, it may
well prove to be the preferred solution. This style of solution is based on the same ideas as a
token passing ring, as used in a local area network. A token is passed from bus master to
bus master. This token confers the right to make use of the bus. If bus usage is not
desired, the token is passed to the next bus master. If use is desired, the token is held until
the bus master is finished with the bus, then passed to the next bus master. In order to
assure that there are no race conditions some means has to be available to indicate a safe
point at which to change between these two modes of operation. Figure 5.2 shows how
such an arbitrator would fit between the bus master and the bus.

![Distributed Bus Arbitrator Diagram](image)

**Figure 5.2 Ring Based Bus Arbitration**

The **FREE-IN** and **FREE-OUT** lines carry the token which grants permission to
use the bus. The **FREE-OUT** line from the last bus master is the **FREE-IN** line of the
first. The **CHECK-IN** and **CHECK-OUT** lines carry some form of clocking signal
which provides a safe period in which to perform any internal operations. The **BID** line
indicates to the arbitrator that the bus master wishes to use the bus. The GRANT line carries a pulse, from the arbitrator to the bus master, indicating that it has been given the right to use the bus.

![Diagram of Distributed Bus Arbitrator When Bus Not Needed](image)

**Figure 5.3 Distributed Bus Arbitrator When Bus Not Needed**

One problem with this form of bus arbitration is that each arbitrator introduces a delay in the passage of the FREE-IN to FREE-OUT signal. If a particular bus master does not need to use the bus, the arbitration circuit should treat the FREE-IN and FREE-OUT lines as one single continuous wire, as shown in figure 5.3.

![Diagram of Distributed Bus Arbitrator When Bus Needed](image)

**Figure 5.4 Distributed Bus Arbitrator When Bus Needed**

The circuitry used will obviously introduce more delay than a wire would. The simplest circuit that provides the desired result is an AND gate with the two inputs FREE-IN and the inverse of the BID line. If the bus master does not need to use the bus, BID will be false, and there will be one gate delay in the propagation of the signal from the FREE-IN to the FREE-OUT lines. If the bus master needs to use the bus, the arbitration
circuit should, effectively, be as shown in figure 5.4. When a pulse arrives on the FREE-IN line, it causes a pulse to leave on the GRANT line.

The passing of the pulse from FREE-IN to GRANT can also be accomplished by an AND gate. The two inputs to the AND gate are FREE-IN and BID. In order to produce a pulse on FREE-OUT after having used the bus, a falling edge of the BID line can be used to trigger a pulse generator to provide a pulse on FREE-OUT. If it is assumed that BID never changes state at the wrong time, the circuitry needed in the bus arbitrator is as shown in figure 5.5.

![Figure 5.5 Bus Arbitration Circuitry With Assumptions](image)

If the BID line is never asserted while FREE-IN is high, no problems will ever arise. When a pulse arrives on FREE-IN, the two AND gates will already be conditioned to either accept use of the bus, or pass it on to the next bus master. The BID line will never fall while FREE-IN is asserted since this is only ever done when relinquishing the bus, implying that the pulse was previously captured. The pulse generator will never introduce a pulse while one exists. All that remains, is to assure that the BID line is never asserted at the wrong time. Here the CHECK-IN and CHECK-OUT lines become important.

As these two lines are one continuous wire, it can be referred to as CHECK from this point on. It carries a pulse which precedes the pulse on the FREE-IN line. The time between the rising edge of the pulse on this line, and the pulse on the FREE-IN line must be great enough, to allow the external BID signal to be latched locally. When the external BID is removed, so is the internal BID, to correctly preset the two AND gates, and to clock the pulse generator. Removal of the internal BID should, rather than triggering the
FREE-OUT pulse generator, trigger the CHECK pulse generator. The FREE-OUT
generator can then be triggered after an appropriate delay. The full circuitry is seen in figure 5.6.

Figure 5.6 Bus Arbitration Circuitry Without Assumptions

The problem with the solution given is that the CHECK line carries a pulse. If there
is no potential bus master which wants to use the bus, at the time the pulse is present, no
potential bus master will ever catch the FREE-IN pulse, because there will never be
another pulse on the CHECK line. Provided there is always a potential bus master which
wants to use the bus when the CHECK pulse is generated all works correctly. The
simplest approach is to assure that this is always the case.

Assume that there is a potential bus master which constantly wants to use the bus. It
finishes with the bus very swiftly. As soon as it is granted the bus, it drops its BID line,
then raises it again before the pulse generator has even completed its task. The bus
arbitration circuitry for such a bus master can be drastically simplified. Given one of these
simplified arbitrators, an extra input can be supplied which can be used at initial power on
of the machine to start the whole cycle. One such device on the bus will serve as a pulse
regenerator and as the initial generator as well. The circuitry for this device's “bus
arbitration” is seen in figure 5.7.

Should there be a bus master which wants to use the bus, the regenerator will still
catch, and delay, the two pulses. If it is known that some bus master will catch the
CHECK signal, there is no need for the regenerator to be involved. Whether or not this is a

179
reasonable improvement remains to be seen. It may turn out to be of little value. Such an improvement may even introduce a negative gain. If the amount of time saved is, say 15 nano-seconds, but to get this saving, the extra complexity in the bus arbitration logic introduces a 1 nano-second delay, a machine with sixteen bus masters would run faster without the improvement.

The time between the CATCH and FREE-OUT signals has to be great enough to assure that the next bus master in the sequence has a consistent signal on its internal BID line. This is a major component of the total arbitration time. Should the next bus master in sequence not require the bus, the delay is excessive. The delay in the transmission of the signal from FREE-IN to FREE-OUT of the next bus master will increase the time difference for the bus master which follows it. Careful timing of the lowering of the BID signal to the arbitration circuitry can remove some of this overhead. The discussion made it appear as if the BID signal is lowered when the bus master is finished with the bus. The signal can be lowered slightly earlier. The next bus master to use the bus will take some time to respond to being granted the bus. All that is required is to assure that two bus masters are not attempting to provide signals on any lines at the same time. The situation is analogous to precision driving demonstrations on figure eight tracks. Two cars can be in the intersection at the same time, provided they are not attempting to use exactly the same surface area.

Figure 5.7 Pulse Regeneration and Initialization
By introducing an **IN-USE** line, the arbitration can proceed concurrently with bus use. A **bus master** which has been granted use of the bus will refrain from using it until the **IN-USE** signal is low. When the signal is low, it places a high signal on the line, and commences to use the bus. After a suitably short delay it can safely drop its **BID** line, allowing arbitration to pick the next **bus master**. The arbitration delay is reduced to the propagation delay of the **IN-USE** signal.

The arbitration delay will be almost completely hidden by the time the bus is being used. The only time this is not true is when the bus is very lightly used. If the bus is underutilized, the signal may have to pass through many **AND** gates, and the regeneration unit, before reaching the next requesting master.

The delays of the **AND** gates can be anything from a few nano-seconds with common components, to much less than one nano-second if **GaAs** components are utilized. This is one place where the choice of an appropriate technology may be important. The circuitry is sufficiently simple that the cost of the highest speed components available should not greatly influence the total cost of the machine, however lower speed, and cost, components can be used.

Given that the ability to obtain the use of the bus as a **bus master** has been covered, now is the time to consider what to do with it. The two basic operations required are giving information to some device, or requesting information from some device. Giving, or writing information is the simplest case.

To write information on the bus, the **bus master** must place both the address, and data in question, on the appropriate bus lines. As well as these two pieces of information, the number of used data lines has to be indicated. Given 128 data lines, a bus master can write 16, 32, 64 or 128 bits of information. Two signal lines indicate the size of the data, and these can be set at the same time as the address and data lines. The signal which indicates that the requested operation is a write has to be set. After giving time for the signals to settle, the bus master raises a control line which indicates that the address lines are stable and the devices can safely check to see if they are the device being addressed.

There are now three situations possible. First, the addressed device exists, and accepts the data. Second, the addressed device does not exist. Third, the addressed device exists, but is busy and cannot currently accept the data. For both the first and third cases,
the addressed device can return a status indication. For the second there is no device to do so. For the sake of argument let it be assumed that such a signal can be generated. If this is true, the handling of the bus by the **bus master** is equivalent, whatever the status signal may be. The different signals indicate what the future operation of the **bus master** should be, but have no affect on the current use of the bus. The timing diagram for a write operation on the bus, after the grant has been accepted, is shown in figure 5.8.

![Timing Diagram](image.png)

**Figure 5.8 Bus Write Timing Diagram**

At time **t1** the **GRANT** pulse has arrived, and the address, data and control lines can be set as appropriate. At time **t2** the signal that indicates that the devices may safely check for their addresses is raised. The time between **t1** and **t2** must be long enough to assure that all devices have properly interpreted the address lines. At time **t3** the **RESPONSE** line raises to indicate that the write operation has terminated. The time between **t2** and **t3** is dependent on the speed of the device in question. At time **t4** the **CATCH** line is lowered to indicate to the device that the response has been received. The time between **t3** and **t4** should be as short as possible. At time **t5** the signals on the address, data and control lines are allowed to float. The time between **t4** and **t5** must be long enough for all devices to have detected that the **CATCH** signal has gone low. At time **t6** the acknowledgement that the device is finished arrives. The time between **t5** and **t6** is device dependent. At time **t7** the **IN-USE** signal is lowered to allow the next **bus master** to use the bus. The time between **t6** and **t7** should be as short as possible.
If the RESPONSE signal indicates that the operation was successful, the bus is no longer needed for the required operation. If the RESPONSE signal indicates that the device was busy, the BID line is again raised and, when granted the bus, the bus master can repeat the operation. If the RESPONSE signal indicates that no such device exists, the operation has terminated unsuccessfully. The generation of a successful or delayed RESPONSE is obvious. The one which requires attention is the one which indicates non-existence.

It would be convenient if it were impossible to use an address which does not correspond with a device. This is, in general, not possible. Consider the library code segment which is available to all processes. If the area is not fully populated, to prevent invalid addresses, the total available address space in the segment would have to be restricted. Given that a FORTRAN program expects a FORTRAN run time support area to exist at some predetermined location in this segment, and a COBOL program expects a corresponding COBOL run time support section at some other location, one has to come before the other. If a third language is considered, there is the possibility of a “hole” in the address space. Given that there are other areas of this address space where common routines can be found, the problem is obvious. To assure that there cannot be invalid addresses on the bus, an arbitrary number of segments would have to exist, one for each section which is addressable by a given program. Ignoring even this argument, consideration has to be given to how the initial configuration of the machine is determined by the kernel.

Placing a “stub” device in the machine in place of a device which does not exist is a possibility. The physical location of a non-memory device does not need to be permanently recorded anywhere. These devices can be all “pushed” to one end of the physical address space reserved for such devices, and a “stub” device can cover the rest of the address space. If these devices are given sections that are 65,536 units in length, there are 16 bits of address which need be covered. A “stub” device for a 4-bit address is shown in figure 5.9.

One quarter of the total 32-bit address space is safe from invalid addresses. One half is RAM and can be covered in exactly the same manner. Three quarters of the problem is solved by the introduction of two simple but slow devices. The speed is not important since they will only be addressed when the kernel of the system is initially determining the configuration. The problem area left is that quarter of the address space which contains the library routines.
The basic problem with the library area is that compiled programs have to "know" the addresses of the routines within the specific library areas of interest, and the library routines have to "know" the same addresses. Given a convenient, and efficient, way to "find out" the addresses in question, there is no need to "know". A common solution is to store the addresses of the functions in a table and call the functions indirectly rather than directly. This reduces the "know" list down to one, the address of the table. Assume that each of the library areas is a fixed length, and the first addressable unit in each contains a unique numeric identifier of that area. For example, the FORTRAN run time area can be given the number "1", COBOL the number "2", etc. At initialization time the kernel steps through these areas, reading the numeric code for each. The address of the area is stored in a table in the kernel's data segment, indexed by this unique code. When a FORTRAN run time routine is called the appropriate entry in the kernel's table points to the area in question, which starts with the table of addresses. This table of addresses contains the offsets of the routines within the area. If the manufacturers of a specific library area feel that this is too much overhead, they can provide the table area in modifiable non-volatile memory, and a utility to allow either customers or dealers to "adjust" the contents of the table to a specific area. This would reduce the overhead down to two loads, one of which is from a cached area and would probably not require an actual memory reference. Going further, a register can be dedicated to holding the table entry from the kernel's table, reducing the address calculation to one load from a cached segment. Such indirect function calls are not uncommon in programming languages. It should also be noted that the actions of the kernel
do not in any way force specific solutions to be used. It provides a basic means of supporting a solution.

The result of the previous discussion is that a "stub" device can also be used to cover the part of the library segment which is not populated. The address space of the machine appears on the bus as six sections as shown in figure 5.10. Three of them are "stub" devices which assure that all addresses on the bus correspond to physical devices.

![Figure 5.10 Address Space Divisions on Bus](image)

For read operations, the same problems would apply as for write operations, the difference being that the device places the data on the bus rather than the bus master. The timing diagram for read operations is seen in figure 5.11.

![Figure 5.11 Bus Read Timing Diagram](image)

This bus provides the support for multiple bus masters, be they occasional or constant users of the bus. The distributed arbitration based on a token passing ring concept allows the number of bus masters to vary as needed. Two signals are required on the bus for this arbitration scheme. Each bus master not requiring the bus introduces a small time penalty in
the arbitration. The use of one other line allows arbitration to take place concurrently with
bus use. The introduction of three “stub” devices assures that all possible addresses on the
bus are “valid” and there is no need for watch-dog timers. The response returned from
devices can indicate a status enabling appropriate further actions by the bus master.

Section 5.2 The Processor Board

The processor board is quite interesting. It has to act as a bus master, and must accept
information given to it, in effect, as a memory board. When a processor is to be given a
new process to execute, that processor has to be “interrupted”, and given a description of
the new process. Rather than attempting a slow and laborious argument to lead to the final
definition of what the processor board will be, the final design is first given, followed by
the reasons which lead to it. This is possible as much of the reasoning has been covered
earlier.

The final appearance of the processor board is seen in figure 5.12. Each of the named
components is worthy of detailed discussion in itself. To get a basic picture of what is
happening a quick run through of what happens in general is valuable.

After reset the PROCESSOR waits for a pulse on the FIFO FULL line, and sends
a pulse out the FREE line to the MASTER unit. All CO-PROCESSOR units are idle,
waiting to be addressed. The CACHE is set to contain nothing, and awaits the first request
from the PROCESSOR. The MMU provides direct mapping of addresses, although this
is not important. The MASTER interface on the BUS, on receiving a pulse on the FREE
line, writes the bit pattern which defines the processor type to an address dedicated to the
processor board in question. The DEVICE interface is set to accept any values written to it.
The whole board is waiting for values to be written to its DEVICE interface on the BUS.

As each value arrives at the DEVICE interface, it is passed to the ROUTE &
count unit. Here a simple octal counter is incremented. Appropriate bits of each value
are routed to the CACHE and MMU. At the arrival of the eighth value the FIFO FULL
pulse is generated, and the DEVICE interface is told to no longer accept any values. The
arrival of the FIFO FULL pulse to the PROCESSOR causes it to generate the LOAD
pulse, which goes to the CACHE unit. The CACHE unit uses the bits from the ROUTE
& COUNT unit to determine which stored values it may retain, then passes the LOAD
pulse to the MMU. The MMU uses the values passed to it by the ROUTE & COUNT
unit to set its internal mapping registers. The pulse from the **CACHE** causes the **MASTER** unit to the **BUS** to again write the processor descriptor to the processor specific location, with the lower bit of the address set to a one, to indicate that the processor has accepted the process given to it. This **LOAD** pulse is also propagated to the **DEVICE** interface to inform it that it may again accept values. At this point the processor can run the process it was given.

![Diagram of the Processor Board](image)

*Figure 5.12 The Processor Board*

Should the process being executed voluntarily relinquish the processor, the **PROCESSOR** saves the state of the process, pulses the **FREE** line, then waits for the
FIFO FULL pulse. The FREE pulse has the MASTER announce that the processor is free.

Should the ROUTE & COUNT unit be given a full process description before the current process relinquishes the processor, the FIFO FULL pulse will force the PROCESSOR to save the state of the current process, then pulse the LOAD line.

The processor board goes through simple state transitions. These internal transitions are shown in figure 5.13, while the apparent transitions to components outside the processor board can be seen in figure 5.14. In the previous chapter the kernel's view of the states of a processor were shown in figure 4.28. Looking at these three different views of a processor's state, it can be seen that there is no direct one-to-one correspondence. The kernel has to deduce the processor's true state, as far as it matters, from the external appearance of the state.

Figure 5.13 Processor Board Internal State Transitions

```
  IDLE  
  |     
  |     
  |     
FIFO FULL  
  |     
  |     
FREE     
  |     
  |     
RUNNING 
```

Figure 5.14 Processor Board External State Transitions

```
  IDLE  
  |     
  |     
  |     
ODD ADDRESS WRITE  
  |     
  |     
EVEN ADDRESS WRITE  
  |     
  |     
RUNNING 
```
5.2.1 The MASTER Unit

The MASTER unit, as well as being a bus master interface to the BUS, is in charge of indicating the state of the processor to the components outside of the processor board. It does this by writing a specific value to a specific location. This action is initiated whenever it receives a pulse on either the FREE or LOAD lines.

The value it writes consists of the bit pattern which describes the processor board. This bit pattern was discussed in vague terms previously. What it consists of is two 16-bit values. The most significant 16 bits specify the exact PROCESSOR type that is installed. The least significant 16 bits are taken from the contents of a bank of switches on the processor board. Each of these switches is assigned to a specific type of CO-PROCESSOR which can be installed on the processor board. The PROCESSOR type bits are used by the kernel of the system to reduce the set of processors in the machine to that set which contains those PROCESSORS which can correctly execute a specific process. The CO-PROCESSOR bits are used to attempt to choose the “best” of this set for the process in question.

The address it writes to is, like the value, composed of two 16-bit values. The most significant 16-bits consist of the value $C000 which addresses the first device in the machines device address space. The lower 16-bits are taken from another bank of switches, and the LOAD or FREE line. The least significant bit is a zero, if writing was initiated by a pulse on the FREE line, or a one if the LOAD line was pulsed. The other 15 bits which come from the bank of switches contain the number of the processor board. Every write which is initiated by the MASTER unit tells the external components “what” the processor is, “which” processor it is, and “why” it is making such a statement.

When neither the FREE or LOAD lines have been pulsed, the MASTER unit passes addresses and data between the MMU and BUS in a transparent manner.

5.2.2 The DEVICE Unit

The DEVICE unit is, to other users of the BUS, a piece of memory. It operates in two states. If a pulse appears on the BUSY line, it becomes a very slow memory component. When it is addressed, it responds by informing the user that it is busy and that the request should be tried again. When a pulse appears on the LOAD line, it becomes a
normal speed memory device. It accepts all requests which are addressed to it, and passes these requests to the ROUTE & COUNT unit.

The address it responds to has the upper 16 bits hard-wired to $\text{C001}$. All processor boards reside in the address space reserved for the second device. The next 15 bits are taken from the same bank of switches as the address used by the MASTER unit. The least significant bit of the address is ignored.

The DEVICE unit will be addressed when the kernel wishes to dispatch a process to this processor board. In figure 4.13 the shared segment descriptor which is stored by the kernel was shown. The address and length word is the value written to the DEVICE unit. Eight words in that format provide all the information necessary for the processor board to execute the process in question.

When the DEVICE unit latches the data it passes a signal to the ROUTE & COUNT unit which takes the data provided, and will pass back a pulse on the BUSY line, if all eight words of data have arrived.

5.2.3 The ROUTE & COUNT Unit

The ROUTE & COUNT unit is trivial. The routing is done with no active components. Appropriate lines lead to the MMU or CACHE units. When an arrival signal comes from the DEVICE unit an octal counter is incremented. The signal is passed through to the MMU and CACHE units. What these two units do with the signals, and data provided, will be seen when they are covered. The overflow of the counter causes the FIFO FULL line to be pulsed. This pulse also goes to the DEVICE unit, in the form of the BUSY line. Between the ROUTE & COUNT unit and the PROCESSOR this pulse may be held by any CO-PROCESSOR which has to. The reasons for this will be seen when a general CO-PROCESSOR unit is covered.

These last two units provide the means of assigning a process to a processor. If a processor is idle it places absolutely no load on the BUS. It is now time to turn attention to the rest of the left half of figure 5.12.

5.2.4 The Memory Management Unit

The functionality of the MMU unit has been covered in some detail in an earlier chapter. Here the discussion changes from what it does, to also encompass how it does it.
Looking at figure 5.15, it is easy to see how a logical address is changed to a physical address. The three segment number bits of the address select one of the base address registers, which provides the upper 26 bits of the 32-bit physical address. The 29-bit offset is bitwise ored into the lower bits of the physical address. The use of the buddy system of memory allocation in the kernel guarantees that any non-zero bits in the 23-bit overlap will come from only one of the two operands.

A simple paged scheme requires an adder to generate the address of the stored page table entry in memory. Constant use of the adder can be avoided by cunning caching of previously loaded table entries, but an adder is still required. A normal segmentation scheme requires an adder to generate the physical address of the requested location. This scheme requires no adder to generate physical from logical addresses, and results in a saving of both access time overhead and physical component count.

Looking at figure 5.16, it is also reasonably clear how an offset which is out of range is detected. The limit values stored are values with the most significant bits set to ones, and the least significant bits set to zeros. The buddy system assures that all valid offsets will be below a given power of two. Forcing the least significant bit of the offset to a one assures that invalid addresses will be detected. This is necessary if the specified segment is completely invalid and the offset given has the value zero. Forcing this bit to a value of one avoids any special case handling. It does requires that, for a valid segment, the minimum
amount of memory which can be allowed is two addressable units, which is an acceptable situation.

Another normally expensive aspect of a segmentation system is the comparitor needed to detect if an offset within a segment is valid, required because limits can be any arbitrary value. Using a buddy scheme there is again no need for the adder since the valid limit values nicely match powers of two.

Looking back to figure 5.12, it can be seen that as well as having access to the addresses flowing between the CACHE and MASTER units, the MMU has input provided on LOAD and NEW MAP lines. Figure 5.17 shows a single cell of a single base or limit register.

Under normal operating conditions, the appropriate pair of eight base and limit registers is selected based on the segment number in the logical address. The correct NOW memory cell places its value on the line which goes to the BITWISE OR or BITWISE AND sections respectively. These two register banks operate exactly as one would expect any normal register bank to operate. When a new word of data arrives at the DEVICE unit, and goes through the ROUTE & COUNT unit, the difference from a normal register bank can be seen.

The upper 26 bits of the new word are placed at the head of the base register data bank, and the next 5 bits are used to generate the 29 bits for the limit register bank. A
SHIFT signal is applied to all storage cells. Each NEW memory cell provides its current value to the next NEW memory cell in the chain, and accepts a new value from the previous. The first in the chain gets its value from the arrived data. After eight new words of data have arrived all NEW cells will contain the required data to describe the logical to physical mapping for the next process to be run.

![Diagram of MMU Storage Cell]

**Figure 5.17 MMU Storage Cell**

When the PROCESSOR responds to the FIFO FULL pulse (now it is clear why this pulse is named the way it is), it will generate a LOAD pulse which goes through the CACHE unit to reach the MMU. The LOAD pulse causes the value in the NEW memory cell to be copied to the NOW memory cell. Now all logical to physical mapping will be done with respect to the newly loaded values.

Most of the time needed to load the MMU with new values is overlapped with the time taken to pass the value to the MMU. Should the processor currently be running a process, it can continue to do so while new values arrive. The kernel is free to generate the new values for the MMU at whatever rate is best for it. There is no need to collect them all before sending them to the appropriate MMU.

Handling the situation where the MMU detects an invalid offset needs to be covered. There are two positions possible here. Either nothing is done about it, or something is. If nothing is done, there is no need to detect the situation. Such an approach is not viable since it implies that one process using a faulty program can impact other correct processes. Something has to be done.
Either some sort of exception has to be raised, or not. At first glance raising an exception seems to be the reasonable choice. It would allow a faulty program to be detected at the earliest possible point. The exception will not be raised where the fault occurred, but where it was detected. These faults will tend to occur only in programs written in a language which provides the programmer with direct pointer manipulation facilities. As languages move to higher and higher levels, these pointers become more the responsibility of the compiler and run time support routines than the programmer. Detecting these faults detects problems with the compiler or run time support programs. Advances in program verification also will continue to make such faults less and less likely. There is a growing base for arguing that these faults will “never” happen.

The approach in the MMU leans on these arguments to support its fault handling. If the requested operation is a write, and a fault occurs, it does not pass the request on to the MASTER unit, but rather provides the acknowledgment itself. If the requested operation is a read, a simple value of zero is provided by the MMU, again without recourse to the MASTER unit. Responding with a zero value is reasonable both for data accesses and, as will be seen later, for instruction references as well.

Apart from relying on the arguments of others, there are also further reasons for preferring such an approach. Generating a fault signal implies that the PROCESSOR must be prepared to handle it. This can complicate the PROCESSOR. For programming languages which either define or allow the definition of the handling of various exceptions, trying to do so with hardware generated faults can be deviously difficult. The CACHE unit can also perform write operations in a write-behind manner. This is a preferred mode of operation. When a write is attempted the CACHE acknowledges the write, then attempts to pass the value on. In the meantime it can respond to other requests. By the time the MMU detects the fault all indications of where the process was in its program can have been obliterated. The bottom line is that such faults “never” happen, and when they do, no one is sure what to do about it, and any information apart from the fact that a fault occurred may well be misrepresenting the true problem.

In conclusion, the MMU converts logical to physical addresses with the minimum of hardware and a minimum of delay. Detection of invalid addresses is only used to assure that no faulty program can cause problems with any other process. Switching from one mapping set to another is very fast and simple. Loading the MMU with a new mapping set can overlap with the continued execution of the process which is about to be pre-empted.
5.2.5 The CACHE Unit

The CACHE unit sits between any CO-PROCESSOR units and the MMU. As with all such cache units, its job is to detect which addresses correspond to locations it currently holds, and provide or accept the data in question without passing the address further. For write operations the data at the address must eventually be passed further, but these writes can be delayed. A general discussion of caching need not be given here. Such topics as replacement policies are adequately discussed in numerous places [Smith 78]. What is worth covering are the unique aspects of this CACHE unit.

The discussion in a previous chapter lead to the decision that only half of the address space need be, or should be, cached. Since the distinction is easily made by inspection of the upper two bits of the address, the CACHE unit can swiftly decide when it “does not exist”. Addresses and data for which it is not responsible are passed through in a transparent manner.

That same discussion lead to the conclusion that placing the CACHE unit where it is, because of the nature of processes having disjoint cached modifiable segments, means that a pending write mode of operation is relatively easy to implement correctly. A pending write cache can provide benefits that a write-behind cache cannot. A set of repeatedly modified locations can be held, reducing the final write operations to one write operation per location. Every pending write operation must be eventually completed.

As long as the processor is running a process, pending writes can be supported. When the processor attempts to switch processes, the pending writes have to be written. This is why the LOAD and FREE signals go from the PROCESSOR to the CACHE unit. These lines provide an indication to the CACHE unit that all pending writes must be satisfied. These two signals are sufficiently different that they deserve individual discussion.

If the FREE signal arrives, it does so with no prior indication to the CACHE unit that it will be coming. All pending writes will still have to be handled. A possibly long period of time will be spent satisfying the pending write requests. Voluntarily relinquishing the processor can take longer with pending write requests than if the CACHE unit had performed write-behind operations. This is not important, since the processor is going to be
idle for some period of time. The time saved in getting to the point where the processor is relinquished, will cover any extra time needed for the writing of any pending values.

If the LOAD signal arrives, this extra time taken to satisfy the pending write requests can become important. The processor is not going to be idle after all writes are satisfied. The new process given to the processor may be critical and should be switched to as swiftly as possible. It is not important how far the process being preempted has got, only that the new process be switched to immediately. Fortunately this signal does not come without some prior indication.

As each word arrives at the DEVICE unit, and passes through the ROUTE & COUNT unit, one bit goes to the CACHE unit to indicate which segments can be retained. This means that when the first word of the eight arrives the CACHE unit is given an indication that very soon a signal will appear on the LOAD line. It can start satisfying the pending write requests before the LOAD signal is ever generated. Should there be a few pending write requests outstanding, all of them can possibly have been satisfied before the LOAD signal ever arrives. The only time that pending writes can cause some delay, there is an indication that this will be the case and an attempt to lessen the impact can be made.

Nothing has been said about the order of the eight words used to define a new process to a processor. Having seen the potential saving in making the time between the first and last words being given as long as possible, there is some hint. The kernel "knows" one segment which all processors will be given, the system billboard segment. The previous chapter covered what has to be done to correctly match processes with processors. Early in the algorithm the set of processors is identified. Only later is the exact match determined between processes and processors. At that early point, the kernel can send one descriptor word to each of the chosen processors, and then continue with the matching algorithm. This will provide sufficient time for all CACHE units to clear many of their pending writes.

Neither the FREE nor LOAD signal can be passed on until all pending writes have been satisfied. Passing the FREE signal to the MASTER unit would inform the kernel that the processor was finished with the process before this was true. For the LOAD signal, this is again the case, in the situation where the process was forced to relinquish the processor. The more important reason for not passing the LOAD on, until all writes are complete, is that the MMU will switch its mapping registers the moment the LOAD signal reaches it, meaning that the writes would go to the wrong physical addresses.
The bits given to the CACHE unit from the ROUTE & COUNT unit are pushed through a shift register in the same manner as they were in the MMU. When the LOAD signal arrives the response of the CACHE unit is more complex since, for each segment it is not to retain, it must mark all locations from that segment as empty.

The CACHE unit does not pass the FREE or LOAD signal through the MMU to the MASTER unit until it is completely through accessing memory. This has simplifying implications for the MASTER unit. The MASTER unit does not have to be built to deal with the situation where it will have both a memory access to handle, and a control access to announce at the same time.

In conclusion, the CACHE unit can operate as a pending write cache with all the savings that implies. The arrival of the first of the eight words which define a new process to switch to, also provides an indication that pending writes are to be completed, giving the CACHE a chance to complete all pending writes before the description of the new process is complete. This implies that the only time a pending write mode of operation could have undesirable effects, there is an indication early enough, that the delay in switching can be minimized.

The CACHE unit completes the discussion of the simple components on the processor board. The two remaining units, the PROCESSOR and the CO-PROCESSORS are, by far, more complex.

5.2.6 The PROCESSOR Unit

The components already covered give little hint of the PROCESSOR unit's nature. It has one signal coming in which is, in essence, an interrupt line. It has two signals coming out, one essentially an idle indicator, and the other an end of idle indicator. There are very few commercially available processors which could not be, with a small amount of "glue" chips, made to fit into a definition of a PROCESSOR unit. That is one of the design goals.

To assume that perfection has been attained is not a viable proposition. As better processors become available, it is desirable that they can be utilized with a minimal amount of change. The rest of the processor board provides a minimal environment into which a given processor can be integrated.
The point now is to discuss the processor which will serve as the initial and basic processor for the machine. As the rest of this section unfolds, it will be seen that the ability to attach CO-PROCESSORs of arbitrary complexity means that the basic processor can be simple. The basic processor provides a minimal set of instructions necessary to make the total machine functional.

5.2.6.1 The Instruction Set

The instruction set of any machine reduces to a few classes of operations. There is the set of "pick it up" and "put it down" operations necessary to deal with memory. There is the set of "fiddle with it" operations, to perform useful tasks on the values which were picked up. Finally there are the control operations which allow branches and subroutine calls, and conditional operations which are usually typified by conditional branches. For a given machine, each instruction fits into one or more of these classes. Some machines provide instructions which combine memory accesses with the "fiddle" operations, for example. To be useful, any machine must provide at least one instruction which can serve to provide the facilities needed, in each of these general classes. The processor described here is an attempt to provide as small a set of instructions as necessary. This is not to say that it is a RISC processor.

The connotations of the term RISC are neither well defined, nor widely agreed upon. A small number of instructions does not seem to be a valid means of identifying a RISC processor. There are processors with very few instructions, yet they are not commonly considered as RISC machines. A large number of registers is also not a distinguishing aspect. There are machines with large numbers of registers, which are not considered to be RISC processors. The waters are muddied even further by the fact that RISC has become a "good" word. If, by some stretch of the imagination, a processor can be labeled as a RISC processor, this seems to enhance commercial benefits. This processor is not a RISC processor, it is just simple.

Simplicity is, in itself, a desirable goal, but usefulness is required. This is typified in the methods used to provide memory addresses. Addressing modes can be simple or complex. Two of the simplest modes are to allow only absolute addresses, or to allow only the contents of a register to contain an address. A complex mode could allow multiple offsets to be applied to multiple registers in a convoluted sequence, with automatic modification of register contents. The simplest modes are useful in a number of contexts,
but insufficient in others. The more complex the mode of address calculation, the fewer contexts it can be used in, but the greater the saving in those contexts where it can.

Providing only one addressing mode is possible. The most useful single mode is one which computes an address from the value of a constant offset, and the contents of a register. This provides absolute addressing, indirect addressing, and the ability to access fields of records. More complex modes can be simulated by a sequence of instructions. Such a single addressing mode is not quite acceptable for all uses. Control flow addresses within a single routine, while able to be specified as absolute addresses, would tend to consist of mostly redundant information. A means of providing short relative addresses for jumps, could result in a reasonable reduction in the total size of the instructions for a given function. Calling subroutines also could benefit from a simpler addressing mode since, in general, the address of the subroutine is known before execution time, and there is no need to compute it as the offset from the contents of a register. There appear to be two major areas where addresses appear. There is the address of a piece of data, and the address of an instruction. These are sufficiently different that they should be considered separately.

Given that the address of a piece of data can always be represented as an offset from the address stored in a register, there is a need for only one addressing mode for data. The question remaining is what size the offset should be. If absolute addresses are ever used, it appears that the offset should be 32 bits in length to support full addresses. This is extremely wasteful when addressing the fields of records, or local variables on an activation stack. These offsets are generally very small. The ability to use variable size offsets would support these two conflicting requirements.

Variable sizes of offset usually implies variable sizes of instructions. An instruction which has more than one possible length implies that the instruction must be at least partially decoded before it is completely read. Even with each instruction a fixed size, but all instructions not the same length, identification of the instruction must be partially done before the instruction can be totally read. There is no problem, provided that the identification and decoding of instructions can proceed in parallel with the execution of the previous instruction. This is a common and widely accepted technique. A reasonably large offset should be available as the minimal size of offset, provided this does not impact the size of the memory reference instructions.

For addresses which refer to instructions, the first to consider is the short relative jumps. These form a large percentage of the instructions generated, by most compilers, for
most languages. Both forward and backward jumps are required. This implies that the offset from the address of the current instruction should be treated, in some manner, as a signed quantity. The range of the relative address should also be as large as possible, without impacting the size of the instruction itself.

Addresses used as the destination of longer jumps or subroutine calls are of some interest. The area of the address space which is protected from modification is within the upper half of the address space. This would seem to imply that all addresses of this form would either have to be given as 32-bit absolute values, or some register would have to contain a value with the upper bit set so that smaller values could be used. A reasonably large number of programs are probably not going to require more than a medium number of program specific instructions, given the availability of a large shared library. If some means of specifying an address within the first small amount of the protected area is available, most of the instruction addresses will be able to use this small offset. If a form of the instructions is available which supports full 32-bit addresses, large programs, as well as those which generate executable instructions in the data area, can also be supported.

There is always a need to provide some means of changing the address of the next instruction to be executed, depending on execution history. This is obvious in the context of returning from a called subroutine. It also appears in the area of calling subroutines, when execution time specification of the subroutine is required. This is the typical "indirect" subroutine call. A third area is in the handling of multiple destination branches, which exists in many languages. A FORTRAN computed GOTO or a PASCAL CASE statement requires such an ability.

Leaving address calculation for a moment, the "fiddle" class of operations can be supported by a few, or many operations. Some form of addition is a minimal requirement, along with some means of performing logical operations. If a choice must be made between twos-complement or ones-complement operations, twos-complement seems a reasonable choice. Ones-complement can be easily simulated. If division and multiplication are not available, some form of shift operation is usually necessary.

Conditional execution is usually available as conditional branches. This is sufficient, however not necessarily best. A close inspection of the instructions, generated by many compilers, for many machines, shows that often the conditional branch branches over one instruction. A good machine to consider is the Data General NOVA, which only had a conditional skip facility. If the rather obscure wording can be accepted, this is best summed
up by the phrase: “Any instruction can, conditionally, not be executed.” Given that what is not executed can be a branch instruction, and that the condition on which the next instruction is skipped can be replaced with the inverse condition, conditional branches can be implemented as a subset of all conditional instructions.

These few introductory words out of the way, the discussion is best served by presenting the final instruction set of the processor, and then turning to a justification of that set. This instruction set is summarized in figure 5.18. The basic instruction size is one 16-bit object. Of the nine instructions, five are always this basic size. Two of them may be 16, 32 or 48 bits in length. One may be 32 or 48 bits in length. The ninth instruction is currently undefined as no useful operation has been found for it.

![Figure 5.18 Instruction Set](image)

The first area of interest is in calculation of addresses which are performed at execution time, which is used in three of the instructions. The best way to cover this is to look at one of the instructions and then see how it reappears in the other two.

**The LOAD/STORE Instruction**

The processor works on a register-register model, and requires the ability to both load and store values. The basic word size is 32 bits. Short 16-bit values are also supported. There are two different load and two different store instructions, one each for each size. All discussions of this instruction will be with respect to figure 5.19.
Figure 5.19 The LOAD/STORE Instruction

The first bit of note is the S bit. This indicates whether the value being loaded or stored is a 32-bit value or a 16-bit value. If the S bit is a 1, the value is 32 bits in length. When loaded, a 16-bit value fills the least significant bits of the register, and the most significant bits are set to zeros. The next bit, the w bit, indicates the direction of the operation. If the w bit is a 1, the operation is a load, a 0 indicates a store. The “opcode” of the instruction is, essentially, the most significant three bits of the instruction.

The three bit AAA field specifies the register to be either loaded or stored. The three bit BBB field specifies the register to be used as the base address to which the offset is applied. Here the MM and DDDDD fields come into play.

The offset which is added to the contents of the register specified by BBB is determined by the values of the combined S and MM fields. This is detailed in figure 5.20. With the least significant bit of the MM field 0 the instruction is one unit in length. The offset is stored in the DDDDD field, and the upper bit of the MM field. If the operand to be loaded or stored is a 32-bit object, this offset is doubled. The most significant bit of this offset is considered a sign bit and is used to fill the upper 26, or 25, bits of the full 32-bit offset.

<table>
<thead>
<tr>
<th>Size+Mode Bits</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMM</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>DDDDD*2 (Six bit signed)</td>
</tr>
<tr>
<td>001</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>010</td>
<td>DDDDD*2+1 (Six bit signed)</td>
</tr>
<tr>
<td>011</td>
<td>hhhhhhhhhhhhhhhhh1111111111111</td>
</tr>
<tr>
<td>100</td>
<td>DDDDD*4 (Seven bit signed)</td>
</tr>
<tr>
<td>101</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>110</td>
<td>DDDDD*4+2 (Seven bit signed)</td>
</tr>
<tr>
<td>111</td>
<td>hhhhhhhhhhhhhhh111111111111111</td>
</tr>
</tbody>
</table>

Figure 5.20 Offset Determination
This means of providing for short offsets supports most accesses within records and stack frames. A distinction is made between short and long operands for a reason. If no distinction was made, the short offset would allow addressing from -32 to +31 units around the location pointed to by the base register. This would correspond to 64 short operands, or 32 long operands. Of the 64 addresses possible, 32 of them refer to odd addresses, and cannot be used to access long operands. Making use of knowledge of the size of the operand, both 64 or and 64 long operands are allowed. This is not a new idea and has been found in other machines before such as the Algol machine from Burroughs. The address is treated as the index into an array of objects. There is an implication here for compilers. Local variables within a stack frame should be ordered so that the short operands are closer to where the stack frame register points. This allows short addressing to, potentially, more operands than may otherwise be possible. There is also an implication for language design since, if the fields of records can be re-ordered by the compiler, more fields can also be accessible with short offsets. If the language does not allow automatic re-ordering, the programmers should be made aware of this aspect. Interestingly, the ordering for this machine is reversed to that used by many programmers. There is a tendency to build structures with the largest fields first so that “holes” will not be forced when aligning long fields after short fields.

Given that a short offset is not sufficient, a 16 or 32-bit offset can be used. If the least significant bit of the MM field is a 1, the offset is not short and the least significant 16 bits of the true offset can be found immediately following the basic instruction. If the most significant bit of the MM field is 0, the most significant 16 bits of the offset are zero, otherwise these bits are set from the value following the least significant value. Medium length offsets are not considered signed. Long offsets are fully specified and so contain their own definition of whether the offset is signed or not.

Treating the shortest offsets as signed was decided on after long deliberation. If treated as unsigned, the number of objects accessible would not increase, but would extend further in a positive direction, and would be of benefit for accessing fields of large records. The telling point shows when the implementation of stack frames is considered.

A routine has, in general, a list of arguments, a list of local variables, and a list of arguments being passed to other routines. Given a stack frame organization which uses two registers to bracket the frame of the current routine, allowing a short signed offset supports

203
addressing of all three lists in a space efficient manner. This signing of the short offset is one of the areas where programming language considerations influence the hardware.

Because all addresses are specified as an offset, plus the contents of a register, the accessing of operands at absolute addresses is slightly constrained. Some register must contain some known value to make absolute addressing possible. If some register contains the value zero, absolute addressing is obvious. Any value will do, provided the compiler can “know” what it is. If the operand is at location X, and register Y contains the value Z, the offset is \((X-Z)\) when the base register is \(Y\).

Since the \texttt{LOAD ADDRESS} instruction is almost the same as the \texttt{LOAD/STORE} instruction, it is worth considering next.

\textbf{The \texttt{LOAD ADDRESS} Instruction}

The \texttt{LOAD ADDRESS} instruction is almost identical in appearance to the \texttt{LOAD/STORE} instruction. It is shown in figure 5.21. Where the \texttt{S} bit appeared there is now a fixed bit with the value 0, and where the \texttt{w} bit appeared there is a bit with the fixed value of 1.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure521}
\caption{The \texttt{LOAD ADDRESS} Instruction}
\end{figure}

The derivation of the offset is exactly as it was with respect to the \texttt{LOAD/STORE} instruction. This includes the \texttt{S} bit as well. Since this instruction has an “\texttt{S} bit” with the value 0, the last half of the table in figure 5.20 is never accessed. The hardware which builds the offset can treat these two instructions as the same which implies a simpler instruction decoder.

Rather than fetching the operand from the address specified by the sum of the offset and the contents of the base register, the sum is the operand. That is the reason for labeling this instruction as a \texttt{LOAD ADDRESS} instruction. By far the most popular use of this instruction is as a general “add immediate” instruction. Serious consideration was given to having an add immediate instruction since such an operation is so popular with compilers. If
the destination register was not the same as the base register, it would not be as useful as the provided instruction since it would not cater for the typical $A = B + \text{constant}$ construct which is often encountered. Implementing an add immediate constant instruction the chosen way also means that the hardware to implement the LOAD/STORE can be reused for this instruction. That is why the $w$ bit in the LOAD/STORE indicates a load when it is a 1. It folds exactly into this instruction. If the most significant bit of the instruction is used to indicate to the memory access unit that it is to access memory, these two instructions are, to the hardware, one instruction.

Another instruction is “almost” a load instruction and so will be covered next.

**The FLYING LEAP Instruction**

There always exists a need to support a “computed goto” of some kind. Even returning from a subroutine is, in essence, a computed goto. Many machines have both a computed goto and a return instruction. Simulating a computed goto with a return instruction can be interesting, (the first Port compiler for the Intel 8086, due to a miss-reading of the hardware manual, actually did this.) Similarly, if the return instruction does many things, simulation of the return instruction can also be arduous. The approach chosen here was to implement a computed goto instruction, and to use it as the return instruction.

A complex return instruction was considered but rejected. The more complex the return instruction is, the more silicon it takes to implement, and the more tightly compilers are constrained to “do it the right way”. The major concern was that “the right way”, will not fit all language’s definitions of how a subroutine call and return should operate. Since a complex return instruction is not needed, a computed goto will do. The format of the FLYING LEAP instruction is seen in figure 5.22.

![FLYING LEAP Instruction](image)

Figure 5.22 The FLYING LEAP Instruction

The instruction derives an offset exactly the way the two previous instructions do. It is worth noting that the instruction in figure 5.20 is reduced down to one 16-bit piece. That the offset available is restricted in range, and is always even is of little concern. By far the most common usage of this instruction will have an offset of zero. Only very cunning compilers will manage to make use of a non-zero offset. Not having the offset at all would
require treating this instruction as different from the LOAD/STORE and LOAD ADDRESS instructions. It is just a LOAD ADDRESS instruction, where the destination register happens to be the register containing the address of the next instruction to execute, rather than one of the eight general purpose registers. Slightly devious routing and manipulation of signals within the processor chip can have this instruction treated exactly as a LOAD ADDRESS instruction. Simulating the w bit being a 1, and setting the fourth line used for register selection to a 1, will make this instruction load register nine, the instruction pointer register.

All three instructions are treated by the majority of the hardware as one instruction. Loads, stores, constant additions, and computed gotos have all been covered. Since the computed goto was used to return from a subroutine, the logical instruction to look at next is the subroutine calling instruction.

The CALL/JUMP Instruction

The CALL/JUMP instruction is shown in figure 5.23. This instruction is either two or three units in length. It serves as both a subroutine calling instruction, and as an instruction which can jump to any addressable location.

![Figure 5.23 The CALL/JUMP Instruction](image)

The L bit indicates the length of the instruction. If it is a 1, the address to call or jump to is the concatenation of the two following units. If it is a 0, the single next unit is needed, and the most significant bits of the address are set to \text{1hh00000000hhhh}, with the lower six bits of the instruction forming the lower four bits and the other two bits of the segment specification. There is a short form of addressing to any location within the first 1,048,576 addressable units of any code segment. Given that the code generated for this processor is reasonably dense, and that a message passing operating system tends to result in small programs being used by cooperating processes, most calls and long jumps can be made with the short form of this instruction.
The K bit indicates whether or not the old value of the instruction pointer should be saved in the register specified by the 1RR field. The K bit is zero when the instruction is to be a long jump instruction. Cunning compilers can also use it in certain cases of tail recursion elimination. Apart from these two aspects, this is a basic subroutine call instruction. It appears strange that the register which can be used to hold the return address is restricted to being one of the four registers R4 to R7. When internal instruction decoding is covered the reason will become clear.

The HOP Instruction

As well as a long form of a jump instruction, it is advantageous if a shorter version is available since many of the control flow destinations tend to be quite close to the source locations. This is provided by the HOP instruction shown in figure 5.24.

![HOP Instruction](image)

Figure 5.24 The HOP Instruction

This instruction is a common one. Many machines have a short jump instruction which contains a signed value to add to the instruction pointer. Many of these machines provide a short form which allows eight bits of offset. This instruction allows twelve bits. The extra four bits provide a greater range, so the long form jump instruction will very seldom have to be used.

All five instructions already covered do not deal with any condition codes. The usual place they are encountered is with branches, which logically should be associated with some form of the jump instruction. Conditional operations are handled by a separate instruction in this machine.

The IF Instruction

It is possible, given the capabilities of the FLYING LEAP instruction, to build a machine without any conditional instructions. While a novel exercise, this approach is not worth following. The IF instruction will conditionally skip the next instruction if the condition in question is false. To the programmer the instruction can be presented as executing the next instruction if the test is true. That form of the instruction more easily fits a programming model. The encoding for the instruction is seen in figure 5.25.
Figure 5.25 The IF Instruction

The encoding of the conditional test is stored in the CCC field. Figure 5.26 shows the meanings associated with each possible value. There are two basic versions of this instruction. One deals with the only “condition code” of the machine, the carry bit. The other version deals with the relation between the values of two registers.

The two conditions dealing with the carry are reasonably obvious. They exist, not for direct use from higher level language constructs, but for support routines used to implement higher level constructs. The six conditions dealing with the comparison of two registers are not as obvious.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Skip Next If</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Carry == 0</td>
</tr>
<tr>
<td>001</td>
<td>Left == Right</td>
</tr>
<tr>
<td>010</td>
<td>Left &lt; Right (unsigned)</td>
</tr>
<tr>
<td>011</td>
<td>(Left^Right) &amp; 1 == 0</td>
</tr>
<tr>
<td>100</td>
<td>Carry == 1</td>
</tr>
<tr>
<td>101</td>
<td>Left != Right</td>
</tr>
<tr>
<td>110</td>
<td>Left &gt;= Right (unsigned)</td>
</tr>
<tr>
<td>111</td>
<td>(Left^Right) &amp; $80000000 == 0</td>
</tr>
</tbody>
</table>

Figure 5.26 IF Conditions

What first has to be done is to define what Left and Right in figure 5.26 mean. The Left operand is specified by the AAA and a fields of the instruction. The AAA field indicates which register contains the value in question. The a field indicates whether or not the value of that register should be complemented. The Right operand is similarly defined by the BBB and b fields.

The two tests for less than, and greater than or equal to, are unsigned as this can result in a simpler hardware implementation. Support for signed comparisons is not as important because they are not as frequent. This statement needs some support since it is counter to what is generally considered true.
Tests for equality and inequality are, by their very nature, neither signed nor unsigned. A pattern of bits is either the same as another pattern, or it is not. Loop control variables tend to be counter variables, even in languages such as C where the initialization, incrementing, and testing parts of the loop control need not bear any relation to each other. Most of these loops can, as effectively, be controlled with unsigned as signed counter variables. For the cases where the compiler can deduce that the counter is being decremented, and the test was of the form \( A \geq 0 \), the compiler can reformulate this test as \( A \neq \$FFFFFFF \). In general, if tests for the value of a register with respect to the value zero are supported, and tests for equality are supported, most conditional instructions will be supported. This has been discovered by various RISC processor researchers. Other tests will require an arithmetic operation to precede the test. The question here is whether or not all relations with respect to a value of zero are supported. There is no need to consider any relation other than equivalence for unsigned numbers since if a number is not equal to zero it is, by definition, greater than zero. There are four relations with signed numbers relative to zero which have to be investigated.

Here it is finally obvious, as was hinted when the memory access instructions were covered, that some register should contain the value zero. Checking to see if a register contains a value that is less than zero is covered when the CCC field is 111, and the Right operand is the contents of the zero register, complemented. The same instruction, with the zero register not complemented, covers the greater than or equal case. The other two cases are more interesting, and uncover an interesting aspect of this IF instruction. That aspect is the implementation of logical implication.

If false is assumed to be represented by the value zero, and the instruction XXX is to be executed if \( A \Rightarrow B \) is true, the following three instructions:

\[
\begin{align*}
\text{IF } & A \neq 0 \\
\text{IF } & B \neq 0 \\
& \text{XXX}
\end{align*}
\]

do exactly that. If \( A \) is false, the second IF instruction is skipped, and XXX is executed. If \( A \) is true and \( B \) is false, XXX is skipped. If both are true, XXX is executed. Direct hardware implementation of logical implication tends to be of little use, but it can be easily seen how this can be used to check if a register is less than or equal to zero. If it is equal to zero, the result is true. If it is not, but it has the upper bit set, the result is true. In the example above the second IF instruction is used to check for the less than zero case.
For the greater than case, a solution is to replace the instruction to be implemented with a hop over the instruction to be implemented, and to use the code for the less than or equal to case.

All comparisons with respect to zero are catered for, as well as comparisons for equality between two registers, and comparisons with respect to the carry bit. Interestingly the two conditions indicated by values of 011 and 111 in the CCC field were initially included to support searching for the first or last bits which are either the same or different between two values. This was prompted by a small piece of sideline research which came up. It may be considered a fortuitous accident that one of these has some other use with respect to signed comparisons. A further justification for these two test conditions is that in higher level languages the carry bit is not usually available and many algorithms which would use it if it were available, tend to test the most or least significant bit before removing it from the variable in question.

The ability to complement either, or both, of the register operands in the IF instruction may appear to be a case of overkill. The only apparent use of this feature is to allow it to complement the register which is known to contain zero. It is conceivable to use this to check if a register contained -1, or the largest unsigned number, but that case is seldom encountered. All that can be said is that the complementing of the operands was not so much designed as inherited, as will be seen as the next instruction is discussed.

The **ALU Instruction**

Finally the instruction which does something is encountered. The instruction is the ALU instruction seen in figure 5.27.

```
010001AAAaabsscB BB ALU
```

**Figure 5.27 The ALU Instruction**

As with the IF instruction, the AAA and a fields specify the left operand, and the BBB and b fields specify the right operand. The register specified by AAA is the destination of the operation.

The oo field specifies what the operation is. This is either an addition, bitwise and, bitwise or, or bitwise exclusive or. The cc field specifies how the carry in to the adder is to
be preset. It can either be left as it is, complemented, set to 1, or set to 0. Twos complement subtraction is the addition of the complement of the register being subtracted, with the carry preset to 1. Ones complement subtraction adds the complement of the register being subtracted with the carry set to 0, then adds the zero register to the destination with the carry left from the previous instruction.

The ss field is a post shift indicator, which either leaves the result as it was, shifts it left one through the carry, left twice through the carry, or right once moving the carry to the most significant bit.

Those familiar with some older mini-computers will detect a strong resemblance to those machines. This is not accidental. While appearing contorted a compiler can make use of this instruction to generate results for sub-expressions like A*4+2, or A+B+1, or (A+B)/2 in a single instruction. Multiplication by ten is three instructions, with appropriate bit settings.

Allowing the operands to be complemented, rather than using those two bits to allow sixteen different operations to be supported, was a matter of reasoned choice. Due to required buffering in the hardware, most values are available both in their true and complemented form. A second reason was that this processor should be simple. Having sixteen different operations would mean that the hardware to implement them would have to exist, in a correct form.

The SWITCH Instruction

This instruction is the single concession to the fact that a processor is not an island unto itself. It is similar to supervisor call instructions in other processors. When a process wishes to voluntarily relinquish the processor, usually to have the kernel perform some operation for it, the program being followed will contain a SWITCH instruction. This instruction is seen in figure 5.28.

![Figure 5.28 The SWITCH Instruction](image)

The instruction performs a very simple task. The contents of each of the eight general purpose registers are written to consecutive locations starting at memory location zero. At
location 16, a word is stored which has the value of the carry bit in the upper bit, and the lower sixteen bits set to the value of the switch instruction. The instruction pointer is written to location 18. After all registers have been written to memory, the processor sends a pulse out the FREE line as discussed previously. It then waits for a pulse on the FIFO FULL line. When this pulse arrives, it is passed out the LOAD line, and the processor picks up the ten registers it had previously saved. It then continues with normal execution.

The actions of this instruction are quite similar to those which happen when the FIFO FULL line causes the processor to be interrupted. In such a case the FREE line is not pulsed, and the lower sixteen bits of the word stored at location 16 are all set to ones. After saving all registers, pulsing the LOAD line, and reloading the registers the processor again continues with normal execution.

When the MMU unit was discussed, it was said that if an invalid memory address was given for a read operation that a value of zero was provided. If that access was for an instruction, the processor will assume that a SWITCH instruction was intended. The kernel, when checking the value stored at location 16 to find out what the process wants, will discover that a zero instruction was executed. This can be defined to be not an acceptable variant of the SWITCH statement, and appropriate steps can be taken. While the MMU does nothing with invalid reads, any invalid instruction fetches are trapped.

These eight instructions provide all the capabilities needed to support any general purpose programming language, and to allow switching between processes. The basic instruction packet is 16 bits long. Common condition codes do not exist. Only a carry bit is supported, of all possible condition codes. Rather than a conditional branch, a conditional skip instruction exists. The conditional skip is performed on the result of testing the relation between two values, usually in registers.

5.2.6.2 Internal Instruction Representation

The instructions available are of various, and in some cases variable, lengths. While advantageous when considerations of code density are important, it does lead to complexity in determining where the next instruction starts, and in determining the components of the current instruction. The instruction set has been designed so that the length of the current instruction is fully deducible from bits found in the most significant nine bits of the instruction.
A combinational circuit can produce the indications that either one or two extra 16-bit packets are needed to complete the instruction. If one extra packet is needed, it forms the least significant 16-bits of a 32-bit immediate value. The second extra packet, if required, forms the most significant half of this immediate value. This leads to the first internal representation of instructions.

An internal instruction is fifty bits in length. There is the 16-bit instruction, the 32-bit immediate value, and the 2-bit instruction pointer increment. The 16-bit instruction is exactly as read from memory. The 32-bit immediate value is set either by the contents of extra instruction packets, or the extraction of appropriate bits as discussed previously. If no immediate value is needed, this 32-bit value is set to zeros. The 2-bit instruction pointer increment indicates the number of packets used to complete the instruction, and indirectly indicates the address of the next instruction.

This internal format requires a further instruction decoding to determine the exact instruction to be executed, which is rather wasteful, as some amount of instruction decoding was required to determine that further packets were required. Looking at the instructions, it can be seen that they can be classified into one of four groups. These are the ALU, IF, MEMORY REFERENCE, and FLOW CONTROL groups. The first two do not require the immediate value, while the last two do. This leads to a superior grouping of instructions into one of two distinct groups.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
<th>V</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>- ALU DESTINATION/IF LEFT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>- ALU SOURCE/IF RIGHT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>- ALU/IF COMPLEMENTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>- ALU CARRY PRESET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>- ALU SHIFT POST OPERATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VW</td>
<td>- IF CONDITION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>- ALU OPERATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>- MAU DATA REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>- MAU OPERATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>- MAU BASE REGISTER/CALL SAVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>- IP INCREMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>- UNIT SELECTION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.29 Internal Instruction Format
The ALU/IF group instructions can be completely defined by the least significant fourteen bits of the instruction packet. If the instruction pointer is treated as just a register by the hardware, the MEMORY/CONTROL instructions can be defined by eleven bits. These bits can be extracted, or manufactured, when the instruction is decoded. Carrying the investigation further, the internal representation of an instruction results in a 33-bit internal instruction with a 32-bit immediate value. The definition of the 33-bit internal instruction is shown in figure 5.29.

For discussion the bits in each field are numbered from left to right, starting with 0. The internal operation of the processor can be defined by the following algorithm.

1 Obtain the new instruction to be performed. If the flag indicating that the instruction is to be skipped is set, set the P field to contain only zero bits.

2a Set the carry bit of the adder for the instruction pointer to the value of Q₀, and the increment to all zeros, except for the two least significant bits taken from Q₁ and Q₂.

2b Indicate that the incrementing unit is to perform its task.

2c On completion, if S₀ is 0, store the incremented value of the instruction pointer back into the instruction pointer register. If P₃ is 1, store the incremented value of the instruction pointer into the register indicated by the R field.

3a Pass the contents of the register specified by the R field to the memory access unit. Pass the immediate value to the memory access unit. If S₂ is 0, pass the value of the register specified by the T field to the memory access unit. Pass the S field to the memory access unit.

3b If P₂ is 1, indicate that the memory access unit is to perform its operation.

3c On completion of the memory unit operation, if S₂ is 1, store the value returned by the memory unit into the register specified by the T field.

4a Pass the contents of the registers specified by the Y and Z fields to the ALU. Pass the contents of the U, V, W and X fields to the ALU to be used as an operation code.

4b If P₀ is 1, indicate that the ALU is to perform its operation.

4c On completion of the ALU operation, store the value returned by the ALU into the register specified by the Y field.
5a Pass the contents of the registers specified by the Y and Z fields to the IF unit. Pass the contents of the VW and X fields to the IF unit to be used as an operation code.

5b If $P_1$ is 1, indicate that the IF unit is to perform its operation.

5c On completion of the IF unit operation, set the skip flag to the result returned by the IF unit.

The above algorithm handles all except the SWITCH instruction which requires special attention. The SWITCH instruction essentially replaces steps 2 through 5 with the generation of the FREE signal, simulated stores to dedicated locations, the generation of the LOAD signal, and finally the simulated loads from the same dedicated locations.

Steps 2, 3, 4 and 5 can all be done in parallel. If all units are assumed free at the start of an instruction, and become busy as dictated by the bits in the P field, the instruction has been completed by the time all units are again free. It is essential that the storage of results from the units happen after the units have been given their operands. For example, for the CALL instruction it is important that the value of the instruction pointer to be incremented be the old value, and not the one which is encoded in the immediate value.

There is a one to one correspondence between the external and internal instructions. If the JUMP version of the CALL/JUMP instruction is treated as distinct internally, the internal coding of the eight instructions becomes the nine internal instructions seen in figure 5.30. The "+" at the end of some instructions indicates that the 32-bit immediate value is used. Those bits within the instruction which are not needed are left as blank values. In general the bits of these instructions are extracted from the external instructions and are labeled with the same labels as were used in figure 5.18. While figure 5.30 presents the internal instructions as nine different instructions, by inspecting the P field of the instructions it can be seen that these reduce down to five.

The ALU instructions form one of the five basic instructions. The IF instructions form a second basic instruction. The SWITCH is given a classification of its own due to its uniqueness. The jump instructions and the memory reference instructions are all grouped into one basic instruction. The CALL is given a special classification, because of its unique aspects. Further discussion requires the definition of the exact meaning of the Q field. The instruction pointer is incremented by a dedicated adder circuit. The most significant bit of the Q field is the value to set the carry-in bit to. The two other bits of the Q field define the two least significant bits of the value to be added to the instruction pointer. For the LOAD
ADDRESS and LOAD/STORE instructions these two bits are shown as \( yy \) to indicate that they are computed from the external instruction rather than extracted.

If one further fact is revealed, the reason for the apparently sparse instruction representations will become clear. A small number of instructions are cached within the processor chip, in their internal format. The caching is done so that for small loops there is no need to access memory for the instructions of the loop. The caching is done in an internal format so that there is no need to decode the instructions within these same loops.

Consider the loop that would be required for the implementation of a multiply function, given that the basic machine does not have a multiply instruction. The multiplier is passed in register 1, and the multiplicand in register 2. The return address is
passed in register 1, and the multiplicand in register 2. The return address is saved in
register 5. The result is returned in register 1. Figure 5.31 gives the machine code and
assembler code for the subroutine. This is not the full version of a multiply subroutine as
such things as detecting the shortest of the two to use for the bit test etc. would clutter the
example. It is assumed that the subroutine is stored starting at location $C0000000$.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0000000</td>
<td>2301</td>
<td>R3 = R1;</td>
</tr>
<tr>
<td>C0000001</td>
<td>7141</td>
<td>R1 = 0;</td>
</tr>
<tr>
<td>C0000002</td>
<td>0B18</td>
<td>loop: if( R3&amp;1 ) R1 += R2;</td>
</tr>
<tr>
<td>C0000003</td>
<td>4102</td>
<td></td>
</tr>
<tr>
<td>C0000004</td>
<td>4B20</td>
<td>R3 &gt;&gt;= 1;</td>
</tr>
<tr>
<td>C0000005</td>
<td>0B08</td>
<td>if( R3==0 ) IP = R5;</td>
</tr>
<tr>
<td>C0000006</td>
<td>0105</td>
<td></td>
</tr>
<tr>
<td>C0000007</td>
<td>4202</td>
<td>R2 &lt;&lt;= 1;</td>
</tr>
<tr>
<td>C0000008</td>
<td>1FF9</td>
<td>IP = loop;</td>
</tr>
</tbody>
</table>

Figure 5.31 Multiply Subroutine

Figure 5.32 gives the machine code and internal representation for this subroutine.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
<th>V</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>IMMEDIATE</th>
<th>EXTERNAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00000000</td>
<td>2301</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000000</td>
<td>1141</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000000</td>
<td>0105</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000000</td>
<td>4202</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000000</td>
<td>0105</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000000</td>
<td>C0000002</td>
</tr>
</tbody>
</table>

Figure 5.32 Multiply Subroutine Internal Representation

Inspecting the internal representation, it can be seen that the ALU and
MEMORY/CONTROL instructions are represented by fields that tend to "miss" each
other. For suitable pairs of instructions two internal instructions can be "folded" into one. If
it is possible to detect these "folds", the stored internal representation of the multiply
subroutine would be as seen in figure 5.33. The second instruction has been "folded" into
the first, and the ninth into the eighth. The second and ninth instructions are stored, it is just
that, unless there is a branch to those locations the instructions stored there will never be
executed. The meaning of the first “folded” instruction is: take the value of register 1, and add to it the constant value zero, then store this into register 3; while at the same time, take the values of register 1 and register 0, obtain the result of a bitwise and of these two values, and place it back into register 1. The second folded instruction adds register 2 to itself, while at the same time storing the immediate value into the instruction pointer register. The first “folded” instruction saves very little but the second, being within the loop, results in one less instruction for each execution of the loop.

<table>
<thead>
<tr>
<th>INTERNAL REPRESENTATION</th>
<th>EXTERNAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100 101 001 0010 0011 11 00 00 01 001 001 00000000 2301 7141</td>
<td></td>
</tr>
<tr>
<td>01000 100 000 0000 0000 00 00 11 00 011 000 00000000 0B18</td>
<td></td>
</tr>
<tr>
<td>10000 100 000 0000 0000 00 00 00 00 000 010 00000000 4102</td>
<td></td>
</tr>
<tr>
<td>10000 100 000 0000 0000 00 00 11 00 00 011 000 00000000 4B20</td>
<td></td>
</tr>
<tr>
<td>01000 100 000 0000 0000 00 00 01 00 011 000 00000000 0B08</td>
<td></td>
</tr>
<tr>
<td>00100 001 101 0010 1001 00 00 00 00 000 000 00000000 0105</td>
<td></td>
</tr>
<tr>
<td>10100 101 000 1010 1001 00 00 00 00 010 010 C0000002 4202 1FF9</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.33 Multiply Subroutine Folded Internal Representation

Folding two instructions into one is very easy. The binary representation of the second is “ored” into the representation of the first. The instruction pointer increment field was defined as both a carry preset value and as an increment to allow this folding to be done on a bit level rather than on a field level.

Detecting when an instruction can be folded into the previous is relatively easy. The semantics of the SWITCH instruction implies that it can never have an instruction folded into it. The use of the immediate value implies that one of the two instructions must be either an ALU or an IF instruction. The assignment of bits in the internal format implies that the other must be a MEMORY/CONTROL instruction. The semantics of the IF instruction implies that it can never be the instruction into which another is folded. The semantics of an instruction which alters the instruction pointer, such as a HOP or CALL instruction can also not be folded into.

If the previous instruction was an ALU instruction, the instructions capable of being folded are the memory access instructions. If the correct operation of the new instruction would be compromised by the folding, it is not allowed. Similarly an ALU or IF instruction can be folded into a memory reference instruction, if correctness is assured. For notation, let \( X^a \) represent the value of field \( X \) in the previous instruction and \( X^b \) be the value of the \( X \) field of the new instruction. Similarly \( X_y^a \) and \( X_y^b \) represent the value of
bit \( y \) of the \( X \) field in the previous and new instructions respectively. The detection of the ability to fold can be done with the algorithm:

\[
\text{if( } Pa==10000 \text{ AND } P_2^b==1 \text{ AND } R^b!=Y^a \text{ AND } T^b!=Y^a \text{ ) fold = TRUE;}
\]
\[
\text{if( } Pa==00100 \text{ AND } T_0^a==0 \text{ AND (P}_b==10000 \text{ OR } P_b==01000) \text{ )}
\]
\[
\text{if( } S_2^a==0 \text{ ) fold = TRUE;}
\]
\[
\text{if( } S_2^a==1 \text{ AND } Y^b!=T^a \text{ AND } Z^b!=T^a \text{ ) fold = TRUE;}
\]

Basically the algorithm says that an ALU instruction cannot have a memory access folded on top of it, if the memory access instruction requires the result of the ALU instruction. The second half says that a memory reference instruction can only have an ALU or IF instruction folded on top of it, if it is not a control flow instruction, and if the folded instruction does not depend on the results of the memory access instruction. If the previous instruction was a store, there can be no dependency. For load operations the register that is the destination of the load must not be either of the two operands to the ALU or IF instruction.

An example which exhibits folding in a better way is that of the subroutine which copies a block of memory terminated by a zero value, a common string copy routine. Figure 5.34 gives the assembly and machine code. The rather odd assembler format on the load and store instructions is the chosen means of indicating that a short value is being loaded or stored. As with the previous example, the subroutine is assumed to be loaded at location $C0000000$.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0000000</td>
<td>A301</td>
<td>loop: ( R3 = R1[0]&amp;$FFFF );</td>
</tr>
<tr>
<td>C0000001</td>
<td>7108</td>
<td>( ++R1 );</td>
</tr>
<tr>
<td>C0000002</td>
<td>8302</td>
<td>( R2[0] = R3&amp;$FFFF );</td>
</tr>
<tr>
<td>C0000003</td>
<td>7208</td>
<td>( ++R2 );</td>
</tr>
<tr>
<td>C0000004</td>
<td>0B28</td>
<td>if( ( R3!=0 ) ) ( IP = \text{loop} );</td>
</tr>
<tr>
<td>C0000005</td>
<td>1FFA</td>
<td></td>
</tr>
<tr>
<td>C0000006</td>
<td>0105</td>
<td>( IP = R5 );</td>
</tr>
</tbody>
</table>

**Figure 5.34 String Copy Subroutine**

Figure 5.35 gives the machine code and the internal representation. It is worth noting that internally the machine appears to have automatic incrementing base registers. The value is loaded or stored at the same time as the base register is being incremented. It should be pointed out that the incrementing of the base register is being done by an ALU instruction.
In general, adding or subtracting one from a register can be done either as an ALU or LOAD ADDRESS instruction. The choice made by the compiler as to which is appropriate should be based on knowledge of instruction folding for best results. If the previous or next instruction is to be an ALU instruction, the increment or decrement should be performed by a LOAD ADDRESS instruction. If the previous or next instruction is a memory reference instruction, an ALU instruction is the preferred means. The importance of this decision can be seen in the string copy routine where the loop requires four internal instructions if the ALU is used for the incrementing, but six if the LOAD ADDRESS form is used.

<table>
<thead>
<tr>
<th>INTERNAL REPRESENTATION</th>
<th>EXTERNAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P Q R S T U V W X Y Z IMMEDIATE</td>
<td>A301 7108</td>
</tr>
<tr>
<td>10100 101 001 0011 0011 11 00 01 00 001 000 00000000</td>
<td></td>
</tr>
<tr>
<td>10100 101 010 0001 0011 11 00 01 00 010 000 00000000</td>
<td>8302 7208</td>
</tr>
<tr>
<td>01000 100 000 0000 0000 00 01 01 00 01 00 011 000 00000000</td>
<td>0B28</td>
</tr>
<tr>
<td>00100 001 000 1010 1001 00 00 00 00 000 000 00000000</td>
<td>1FFA</td>
</tr>
<tr>
<td>00100 001 101 0010 1001 00 00 00 00 000 000 00000000</td>
<td>0105</td>
</tr>
</tbody>
</table>

Figure 5.35 String Copy Subroutine Internal Representation

Discussion is all well and good, but a physical implementation is a strong determinant of what is “useful”. If a typical form of implementation is attempted, certain aspects become more difficult than it may have appeared. For example, stating that a LOAD/STORE and an ALU instruction can be folded and executed at the same time, implies that the contents of up to five registers have to be latched onto five separate sets of bus lines, at some point in time. The area of the bus lines is great. When the area needed to form the selection circuitry to support nine different registers being gated to any of five sets of bus lines is considered, the parallel execution of two instructions as one begins to look dubious. What is needed is a different approach to internal organization.

5.2.6.3 Internal Organization

The basic problem with a typical implementation of a processor is that it suffers from exactly the same problems that shared memory programs suffer from. The shared memory is the set of registers. The shared resource either becomes a bottle-neck, limiting the amount of parallelism possible, or expensive methods must be used to increase the bandwidth to the shared resource, so that parallelism is possible.
Rather than attempting to support a shared resource in the hardware it is possible to create sub-processors which communicate between each other. While not completely unique, what is slightly different in the proposed scheme is that the functional units within the processor do not share a bank of registers.

![General Internal Structure](image)

Figure 5.36 General Internal Structure

The data accessing unit and the arithmetic unit do not need to share the same physical registers. All they need to do is assure that, for those registers which matter, they agree on what the contents are. Looking at the string copy example, it is not until the instruction at location $C0000004$ does the IF unit have to “know” what value was loaded by the
DATA ACCESS UNIT with the instruction at location $CO000000. Until that point it can continue "living in the past", with no detrimental effect.

Figure 5.36 gives a rough outline of the structure of the processor chip. The relative sizes of the parts in the figure are not to scale. Certain structures have been shown in expanded size for clarity. The rather unusual object in the centre is at the heart of the matter. It is the communications ring. On this communications ring, all information necessary to the proper functioning of the chip is passed. The three major fields of a message can be termed WHO, WHERE and WHAT.

WHO specifies which units are to receive the message. This is a set of four bits, one for each logical unit. A message can be marked as destined for more than one unit. WHERE is a four-bit field that specifies a register number. It tells the units addressed by the WHO field where it is to place the value found in the WHAT field, which is 32 bits in length.

Each of the units maintains its own consistent internal state. When it changes the value of a register which is known to other units, it must generate a message specifying which units are to be informed, which register has changed, and what the value has changed to.

<table>
<thead>
<tr>
<th>REGISTER</th>
<th>REASON FOR EXISTENCE</th>
<th>UNITS INVOLVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td>STATUS</td>
<td>AIU ✓</td>
</tr>
<tr>
<td>R9</td>
<td>INSTRUCTION POINTER</td>
<td>CLU ✓</td>
</tr>
<tr>
<td>R10</td>
<td>INSTRUCTION</td>
<td>DAU ✓</td>
</tr>
<tr>
<td>R11</td>
<td>INSTRUCTION CONSTANT</td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td>AIU/DAU/IFU COMMAND</td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td>DAU OFFSET</td>
<td>IFU ✓</td>
</tr>
</tbody>
</table>

Figure 5.37 Extra Register Placement

As well as the general registers R0 through R7, each unit has specific registers which are used to control the internal workings of that unit. These now need to be introduced. There are six registers other than the eight general purpose ones. The instruction pointer, R9, and status R8 have been seen before. The others are new. Figure 5.37 lists these
registers, with a short definition and an indication of the units in figure 5.36 which maintain them.

The functioning of the whole chip is based on the correct functioning of the communications ring. The first object worth investigating is the component of each unit which handles the communications ring.

**Communications Ring**

Figure 5.38 shows how each of the bits in the WHERE and WHAT fields of the message pass between the units and the ring. The unit places the WHAT and WHERE fields of the message into the OUT BIT slots. When a message has arrived for a unit, it is found in the IN BIT slots. A message appears on RING INPUT. If the RECEIVE signal arrives at this point, the value from RING INPUT is passed to IN BIT. If the SEND signal arrives at this point, the value from OUT BIT is the value to transmit, otherwise the value to transmit is RING INPUT. Given that the SEND and RECEIVE signals are provided at the correct times the communications ring will function perfectly.

![Figure 5.38 Ring Data Interface](image_url)

The SEND and RECEIVE signals are generated by the section of the interface which handles the WHO field of the message. It is composed of four handlers for bits much like those for the WHERE and WHAT fields, but there is no need for the IN BIT part. The WHO field section has to generate the SEND and RECEIVE signals so it is slightly more complex. It is controlled by four inputs, and goes through four states, seen in
Two of the inputs come from the unit for which it works, while the other two are produced by a function of the message which has just arrived from the ring.

When a message arrives on the ring some number of bits in the WHO field will be non-zero. If the bit which is used to indicate this unit is non-zero, the MINE signal is generated. If no bits are set, the EMPTY signal is generated.

The actions shown in figure 5.39 are reasonably trivial. If a message arrives which is addressed to the unit in question, it removes itself from the list of destinations, and generates the signal to latch the message away, provided there is space to save the message. When there is an empty message that has arrived, noted by the fact that it is addressed to no one, and there is a message to go, it uses the slot to transmit the message.
The **SEND** and **RECEIVE** signals which control the internals of the ring interface are passed to the unit in question, where they serve as **MESSAGE SENT** and **MESSAGE RECEIVED** signals.

Control of when the internal operations of the ring interface happen is provided by a clocking signal on a line parallel to the data lines of the ring. If information is latched into the interface on a rising edge, and latched out on a falling edge, all interfaces on the ring will run in lock step, and not be susceptible to race conditions.

Given that units can successfully communicate, a brief summary of the operations of each of the units shown in figure 5.36 can now be given. The **CLU** controls when an instruction is executed. It assures that all operands needed for an instruction are available before the instruction starts. The **DAU** moves data between the registers and the external store. It performs the address calculation of adding a constant to a register to get the effective address. The **AIU** performs the arithmetic and logical operations on register contents, as well as handling the **IF** instruction. These two separate tasks are combined into one unit due to the tight coupling needed between these two tasks, and because of the similarity of their implementation. The **IFU** obtains the decoded instructions, and performs trivial instruction pointer manipulations internally. Each of these units can now be looked at in some detail.

**Command Launch Unit**

The simplest unit is the **CLU**. This unit is in charge of assuring instructions execute in a correct sequence. It maintains two extra registers beyond the eight general purpose registers. These two registers contain the instruction to be issued next, and the immediate constant which goes with that instruction, if the instruction is a memory reference form of the instruction.

The task of the **CLU** is basically trivial. It obtains a new instruction. It then waits to assure that all registers needed for the instruction are stable. At that point, it sends a message to the other appropriate units, which contains the information they need to perform the instruction. It then repeats the task with yet another instruction.

The **CLU** has a unique definition of the contents of the eight general purpose registers. The **CLU** version of these registers store, not values, but status indications. Each register contains a few bits. These bits indicate the validity of the contents of the registers.
stored throughout the chip, and which registers are needed for correct operations. The best way to gain an understanding of the CLU is to look at the algorithm it follows. This will require specific messages to be sent around the communications ring, and so a means is needed to allow discussion of these messages.

A message on the communication ring can be specified as a triplet, \((X)(Y)(Z)\). The \(X\) is a comma separated list of unit names. These named units are the specified destinations for the message. The \(Y\) is a register number such as \(R3\) or \(R12\). The \(Z\) is either a literal constant specified in hexadecimal, or the contents of one of the registers stored by the unit sending the message. If \(Z\) is the contents of a register, it is specified by the register number such as \(R5\) or \(R11\). A third possibility for the \(Z\) element is designated as ??? which is used when discussing an arrived message, as the value is not yet determined. The fourth possibility is that the value is not important. This is indicated by an empty third element.

At initialization the CLU sends a \((IFU)(R12)()\) message, after marking all registers as valid. It then enters state 0. All transitions and actions shown in figure 5.40.

<table>
<thead>
<tr>
<th>STATE TRANSITIONS</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT</strong></td>
<td><strong>R10</strong></td>
</tr>
<tr>
<td><strong>STATE</strong></td>
<td>0/4</td>
</tr>
<tr>
<td><strong>1</strong></td>
<td>0/4</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>0/4</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>0/4</td>
</tr>
</tbody>
</table>

\(0/4 == 0\) if no register conflicts
\(0/4 == 4\) if any register conflicts

**Figure 5.40 CLU State Transitions and Actions**

A few comments need to be made about figure 5.40. The input \(Rx\) is any register except \(R10\) or \(R11\). The input \(LDR\) is an internally generated signal which indicates that the last dirty register has been made clean. There is a distinction made between receiving input for register \(R10\) and the register “\(R10\)”. This distinction is made internal to the CLU based on the instruction it receives. Looking at the internal instruction formats, it can be seen that one bit in the \(P\) field of the instruction indicates that an immediate constant is needed. That bit is used to identify whether \(R10\) or “\(R10\)” was received. Making the
distinction can reduce the number of messages sent on the ring, as there is no need for a message passing $R11$ if there is no use for $R11$.

Secondly, as noted on the figure, the state 0/4 stands for either state 0, or state 4, depending on whether or not there is a conflict with the registers the new instruction needs and any currently dirty registers. For each register there are three bits. One is the current instruction dirty bit. The second is the next instruction dirty bit. The third is the next instruction needs bit. A conflict exists if any register needed by the new instruction is still currently dirty. While a conflict exists the CLU stalls, waiting until there is no conflict. When there is no conflict it can perform action C, which starts the new instruction and requests yet another. This mechanism can pipeline instructions. The merging of loads and stores with adds, for example, is done in the IFU. Two add instructions cannot be merged because of the overlap in the instruction encoding. While the first add instruction is currently being executed, the second can be cycling on the ring, and a third instruction being prepared before the first has finished. This need not be limited to only having three instructions in preparation at once. If the selected unit is very slow, and instructions can be obtained very fast, a considerable number of instructions can be pending on the communications ring, provided there are no conflicts. This is not likely, but possible.

The existence of two instructions for the AIU, for example, is completely safe due to the above definition. Having the AIU executing one instruction while the second is circulating and the third is being prepared is acceptable. A difficulty arises should more than one instruction for the AIU be cycling. Assuring that no instruction is sent out until all its operands are stable is not sufficient as a proof of correctness, if the order of the instructions can be changed. Consider the three AIU instructions:

\[
\begin{align*}
R1 &= R1 + R2; \\
R3 &= R3 + R2; \\
R2 &= R2*4;
\end{align*}
\]

where the first is being executed and the next two are circulating. If the third is received before the second, the result of the computation will be wrong. Either the order must be preserved, or some means must exist of detecting that placing the third instruction on the ring is not a valid operation.

Ordering the instructions on the ring appears not to be feasible. Even if each message was labeled with a sequence number, the AIU could not accept a message until the ring had cycled completely so that the message with the lowest sequence number could be identified.
This would almost assure that the AIU would appear so slow that multiple messages would be cycling for it.

Building an instruction queue in the AIU, to assure that all messages for it were taken off the ring as they appeared, would occupy a large amount of surface. The queue can be made a safe length, for every instruction modifies at least one register, implying that the queue need not be any longer than the number of registers. Considering that the queue will, in general, be empty, it seems wasteful to build one which is eight instructions in length.

Detection of messages which should not be placed on the ring due to a potential sequencing problem is possible. If there is also a current instruction needs bit as well as the three others already covered, the above example is easily handled. The definition of a conflict is extended to also cover the situation where any register needed by the current instruction is made dirty by the new instruction. The need bits of the current instruction can all be cleared when the last dirty bit of the current instruction is cleared. A minor change to the state transitions and actions shown in figure 5.40 is needed, and the new definition is seen in figure 5.41.

<table>
<thead>
<tr>
<th>STATE TRANSITIONS</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT</strong>&lt;br&gt;STATE</td>
<td><strong>a</strong> = Record R10</td>
</tr>
<tr>
<td>R10</td>
<td>R10i</td>
</tr>
<tr>
<td>0</td>
<td>0/4</td>
</tr>
<tr>
<td>1</td>
<td>0/4</td>
</tr>
<tr>
<td>2</td>
<td>0/4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0/4 == 0 if no register conflicts</td>
<td>a = Set any new dirty/need bits</td>
</tr>
<tr>
<td>0/4 == 4 if any register conflicts</td>
<td>b = Record R11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTIONS</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>c = Merge new dirty/need bits into current dirty/need bits</td>
<td></td>
</tr>
<tr>
<td>Clear new dirty/need bits</td>
<td></td>
</tr>
<tr>
<td>Transmit (DAU)(R13)(R11)*</td>
<td></td>
</tr>
<tr>
<td>(DAU, AIU, IFU)(R12)(R10)</td>
<td></td>
</tr>
<tr>
<td>d = Clear specified current dirty bit</td>
<td></td>
</tr>
<tr>
<td>e = Nothing</td>
<td></td>
</tr>
<tr>
<td>f = Clear current need bits then do action c</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.41 Modified CLU State Transitions and Actions**

If done in software the stall until safe is the looping construct:

```
while( current_dirty&new_need || current_need&new_dirty );
```

In hardware it is waiting for a signal to fall.
This solution of maintaining need bits until all current dirty registers have been validated is a conservative solution. Since there is no stored indication of which needed registers correspond with which instruction, registers can still be marked as needed after this is no longer true. The cost to overcome this is much more than eight more bits to store. Not only do separate bits for each instruction need to be stored, but circuitry would need to be implemented to identify which instruction completed.

The costs of such extra circuitry seem not to be warranted. Instruction fetching has to be faster than the AIU processing for there to be any outstanding AIU messages circulating on the ring. The probability that a conservative approach will incur a drastic performance penalty is extremely small. The effect may be noticeable when executing tight loops which fit entirely within the on-chip instruction cache, but even this is not readily apparent. Considering that identifying an instruction will require at least incrementing the instruction pointer, a reasonable amount of the time spent by the AIU in performing its task will be overlapped with the time used by the instruction fetch unit.

A final thing to note is that action c will only send the \((DAU)(R13)(R11)\) message, if the instruction is going to involve the DAU. Similarly, the DAU or AIU may not be included in the \((DAU,AIU,IFU)(R12)(R10)\) message, if not needed. In figures 5.40 and 5.41 there was a distinction made between receiving input for register R10 and the register \("R10i"\). The same bit in the P field used to identify whether R10 or \("R10i"\) was received also indicates whether or not R11 should be sent to the DAU as R13.

If the immediate constant message is sent to the DAU, the CLU must mark R13 as dirty, due to the fact that two messages must be sent to the DAU. The previous discussion assumed that multiple messages to the same unit could be cycling on the communications ring. This is not the case with the DAU as a destination, since to allow such a situation would be to lose track of which immediate constant went with which instruction. The DAU has to acknowledge that both messages have arrived by sending a \((CLU)(R13)()\) message at the appropriate time.

It may appear that the \((IFU)(R12)(R10)\) message is sent for two separate, yet related reasons. The mere existence of this message being received by the IFU is sufficient to inform it that it can start to supply the next instruction. The contents of R10 is only important in that it indicates whether or not the current instruction is an IF instruction.
When the current instruction is not an IF instruction, the IFU “knows” where the next instruction is. Should the current instruction be an IF, it must wait until the AIU has determined if the physically next instruction is to be executed. Because the IFU was the originator of the current instruction which is in R10, it need not identify an IF instruction by inspection but can base this identification on prior knowledge i.e., “I just gave you an IF instruction, so when you ask for another, I must first wait to find out whether to skip one or not.” To complete this discussion, a more detailed examination of the IFU is appropriate but a few comments on validation are appropriate first.

The CLU is a simple object, interacting with the other components of the chip only by the communication ring. This is also true of the ring interface sections. The same approach used to validate the programs used by communicating processes can be used to validate these hardware components. Each can be isolated and tested exhaustively. Because each component is simple, it is even feasible to attempt a “proof” of correctness of the implementation. From that point the individual units can be treated as “black boxes”, and only their clearly defined interactions need be considered. This is very important for hardware, as fabricating a chip, just to test it, is not an economic solution. Simulation of the chip is currently the only way to “assure” that the implementation is correct. Exhaustive simulation of a complex component can take large amounts of time and, because these components are not trivial combinational circuits, all historical artifacts have to be checked.

**Instruction Fetch Unit**

The IFU does more than “feed” instructions to the CLU. It can perform almost all the instruction pointer modification instructions locally. There is no need for the internal instruction format to carry with it the increment for the instruction pointer. The instruction format passed to the CLU is that shown in figure 5.42. Note that the instruction cache continues to contain the instruction increment, it is just not passed out of the IFU. The P field has also shortened by one bit. The instruction representations shown in figure 5.30 also change to those shown in figure 5.43.
Figure 5.42 Internal Instruction Format

Figure 5.43 Internal Instruction Representations
The **HOP** and the **JUMP** version of the **JUMP/CALL** instruction can be handled completely within the **IFU** and are no longer passed to the rest of the processor. The earlier discussion allowed a **HOP** or **JUMP** to only be folded into a previous **ALU** instruction. By performing the instruction pointer modification within the **IFU**, the time needed for this instruction can also overlap load and store instructions.

The **CALL** instruction has changed its representation subtly. It now is seen by the **CLU** as a load into the appropriate register of an immediate constant which happens to be the address of the instruction which follows the **CALL** instruction. The modification of the instruction pointer is handled as in the **JUMP** case.

The **SWITCH** instruction as seen in figure 5.30 has been replaced with a note in figure 5.43. Earlier the **SWITCH** instruction was discussed in terms of what effect it had, rather than in terms of how it was implemented. There is now enough background to look at implementation.

For the **SWITCH** instruction to complete, two signals have to be generated to the outside world. These are the **FREE** and **LOAD** signals first seen in figure 5.12. Before the **FREE** signal is emitted, all the registers have to be stored. After the **LOAD** signal is emitted, all the registers have to be loaded. This can all be done with a proper sequence of instructions in the internal format. From what has already been covered, all is obvious except for three small points.

The use of internal instructions to have the **DAU** store and load registers **R0** through **R7** is easy to visualize. The **DAU** does not have access to either the status register **R8**, or the instruction pointer **R9**. It is conceivable that the **DAU** could maintain a status register. The **DAU** will never use the status register. The stored status register has to contain the reason for the **SWITCH** in the lower 16 bits, and the carry bit in the most significant bit. The **AIU** will be generating the carry bit value as it changes. The **DAU** will be constantly fed with values to maintain in its status register, but these values will never be referenced. The **IFU** can use a load immediate instruction to get the **DAU** to store the correct value in its status register. It seems excessive to have the **DAU** store the value of the status register since it will, in general, be useless, and when it is needed, wrong.
The same comments can be applied to the instruction pointer R9. The DAU will not be constantly updating R9, the IFU will never “talk” about it. It will have to be updated to a correct value when it is to be stored.

Mixed with the aspect of saving the current state, the loading of the new state also has to be taken into account. There is a complication. While running normally, the operations are in the order FREE, followed by LOAD. At initialization time there is a need to generate the FREE signal, but not store any state, followed by a LOAD signal and the loading of the initial state.

Figure 5.44 Internals of the Instruction Fetch Unit
The chosen solution handles all these aspects in a compact and direct form. To see this, the IFU has to be looked at in more detail. Figure 5.44 shows a diagrammatic representation of the IFU. Sections are not necessarily shown to scale or in the correct positions. From this diagram, the first thing which needs discussion is the **SWITCH INSTRUCTION LIST SEQUENCER (SILS)**.

The **Switch Instruction List Sequencer (SILS)** consists of a small piece of memory, a comparator, and a cycling counter with a period of 23. The memory stores significant bits of the 23 instructions which will be generated whenever the comparator detects that the **FLOW CONTROLLER (FC)** is attempting to load an instruction from location $00000000$. Each time this address is recognized, the counter will indicate the next instruction from the list of 23 which it contains. At initialization the counter is set to contain the value zero. Figure 5.42 showed the instruction internally as containing 32 bits. This is the size stored in the cache. The size of the instruction received by the FC is 34 bits. The cache provided 32 of these, and when it is addressed, provides the other two bits as zero values. These two bits are crucial to the correct operation of the **SILS**.

Figure 5.45 gives the essentials necessary to understand the correct operation of the **SILS**. Every time an instruction is loaded from location $00000000$, the **SILS** will take the currently indexed set of bits A through N, and insert them into the 34-bit instruction and 32-bit immediate constant in the locations shown in figure 5.43. The sequences of bits A through N are shown in figure 5.46. After providing one instruction the counter increments to address the next set of bits in the sequence.

```
<table>
<thead>
<tr>
<th>32-bit Basic Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00A</td>
</tr>
</tbody>
</table>

Special Switch Bits

| H | J |

<table>
<thead>
<tr>
<th>32-bit Immediate Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000000000000000K</td>
</tr>
</tbody>
</table>
```

**Figure 5.45 Switch Bit Insertion Locations**

234
Passing the instructions, formed by applying the bits from the sequences in figure 5.46 to the basic format in figure 5.45, produces all that is necessary to switch contexts, and to load the initial context. The first instruction of this sequence is a SWITCH instruction with two bits set in the S field. When the FC encounters this instruction it waits until it has also received a request for another instruction from the CLU. At that point it will generate a FREE signal if FIFO FULL is not true. When FIFO FULL is true, the FC generates a LOAD signal. At initialization time this instruction provides the FREE pulse needed, then waits for a given context before continuing. It should be noted that the FREE and LOAD signals do not make it out of the processor chip until all units within the chip are idle. This is important when saving the context is considered. After generating the LOAD signal, the FC then attempts to load the next instruction. Since the Q field was 000, the second instruction is also loaded from location $00000000$.

![Figure 5.46 Switch Instruction Bit Sequences](image)

The second through ninth instructions provided by the SILS are instructions which load registers R0 through R7 respectively. The tenth is slightly special. The destination register of the load as R8. This is the status register, and the load will cause the IFU to update its R8, and let the AIU update the carry bit. The eleventh instruction is very special. The destination of the load is R9. R9 only exists within the IFU as the instruction pointer. When this instruction reaches the FC of the IFU, it is treated as any other computed jump is treated. The FC waits until it receives an update of R9 from the DAU, then uses it as the instruction pointer for the next instruction. This new value for R9 will not be a zero, so the SILS will not trap this next instruction load but pass it through. This instruction will be the first of the instructions from the new context which was just loaded. The counter in the SILS will remain stalled waiting for the twelfth instruction to be loaded from location zero.
This twelfth instruction is the first of a 23 long sequence used to save the old context and load a new one. Initialization was the reason why the first instruction was the thirteenth of a complete cycle.

The twelfth through twenty-third instructions in the SILS store the state. Eight of these store R0 through R7. The other four store the status register R8 and the instruction pointer. This is where the two special switch bits of the instruction format are used. If the bit labeled H in figure 5.43 is non-zero, the contents of R8 are ored into the immediate value of the instruction supplied. If the bit labeled J is non-zero, the contents of the hidden register R14 are ored into the immediate value. R14 should be R9, however R9 contains a zero, and is useless. When the FC detects that a switch of state is required, either by instruction or by FIFO FULL, it copies R9 to R14 before setting R9 to zero. More will be said about the handling of instructions by the FC later. These four instructions load the contents of the each register into R0, then have R0 saved in the correct memory location. After the twenty-third instruction is generated, the counter cycles back to zero. From this point on the situation is exactly as it was at initialization time.

As with other parts of this chip, the SILS can be extracted as a part and the implementation exhaustively tested in simulators to assure correctness. The only area which could cause problems is the detection and trapping of references to location zero. Done incorrectly this can add a considerable time penalty to loading instructions. On the other extreme it could lead to race conditions where both the SILS and the cache attempt to supply the contents of location zero.

If the address lines are monitored constantly, any time that they indicate a zero, a signal can be generated which will trap the address available signal from FC. It does not matter if the address available signal is trapped when the address is invalid. This solution will place one gate delay between the FC and the cache which is an acceptable overhead.

The next area of interest in the IFU is the INSTRUCTION DECODER (ID). The ID takes one 16-bit instruction packet and spreads the bits, as appropriate, within the 32-bit internal instruction. The bit layout within the 16-bit instruction packet has been chosen to make this operation trivial. The initial assignment of values to the internal bits of all fields except the P and Q fields can be done by wire routing. This default routing of bits is shown in figure 5.47. To see how the other bits are handled, each instruction type can be looked at individually. To facilitate merging, the identification of the instruction type
indicates which part of the instruction is not needed, and is used to mask off any bits in the unused part.

The ALU instructions are trivial to handle. The P field is set to 1000, and the Q field is set to 100. The difference between the ALU and the IF instructions is that for the IF instructions the P field is set to 0100. The SWITCH instruction is also simple, and sets the P field to 0001 rather than 1000.

The LOAD/STORE and LOAD ADDRESS instructions are now worth considering. They both set the P field to 0010. The upper bit of the S field is set to a 0. The Q field is determined by the offset provided, as specified in the w and MM bits. In fact, all instructions which make use of the immediate constant part of the internal instruction, these two plus the FLYING LEAP instruction, use the same method of deducing the value of the immediate constant. The FLYING LEAP always uses the offset in the DDDDD field of the external instruction, while the other two will most likely use it frequently as well. It is worth while creating a potential immediate constant assuming that the DDDDD field is going to be used. This can be done in parallel with the identification of what the instruction is. For those which do not use an immediate constant the value is not important. For the four of eight cases where this is a valid assumption, the Q field of these two instructions can be set to 001.

![Diagram](image)

**Figure 5.47 External to Internal Default Routing**

In the cases where the immediate constant is not based on the DDDDD field, no time has been lost. Should a second instruction packet be required to complete the immediate constant, the potential value from this computation can be discarded. The second packet is used to set the lower 16-bits of the immediate constant, with the upper 16-bits set to zeros, and the Q field set to 010. Should a third be needed, it goes into the most significant half of the constant, and the Q field is set to 011.
For the FLYING LEAP instruction, since the DDDDD field is the correct value to use for the immediate constant, the P field is set to 0010, the Q field to 001, and the upper bit of the T field is set to a 1, forcing the destination of the instruction to be R9, the instruction pointer. All but the HOP and JUMP/CALL instructions are easily handled. These are more difficult.

For the JUMP/CALL instruction the immediate value is always going to involve at least one more instruction packet. This implies that the most significant part of the immediate value is set to the default $X00X$, with, as mentioned previously, the lower six bits of the instruction packet being used to define the value. If a third packet is needed, it is placed in the most significant half of the immediate constant. The Q field can be set to 01L with the L bit being the L bit out of the external instruction format. The S field is set to the value 10K0, with the K bit taken from the external instruction format. The P field is, of course, set to 0010.

The HOP appears the most difficult. It could be stored as a JUMP version of the JUMP/CALL instruction. Doing so means that the ID would have to contain an adder to compute the true destination of the branch. Given the number of adders already used, introducing another seems excessive. The HOP sets the immediate value to the sign extended offset taken from the instruction packet. The Q field is set to 100. The S field is set to 1000. The reason for this exact format will be seen later, but first, how instructions can be merged together should be covered.

The instruction decoder can continue obtaining and decoding instructions as long as the instruction stream consists of ALU, IF, LOAD/STORE or LOAD ADDRESS instructions. The sequence it goes through is:

1/ Wait until asked
2/ Obtain instruction
3/ Decode instruction
4/ Give to cache to hold
5/ If ok to continue go to 2 else go to 1

The decision of whether to continue or not is based on the type of instruction which has just been processed. Interestingly, the instructions which can be merged into the previous instruction are exactly those which allow the ID to continue loading and decoding instructions. Part of the decision of whether to merge or not, is based on whether or not to
continue loading instructions. Another factor is whether or not the instruction is the first of a sequence. If this is the first in a sequence, merging is not possible. Looking more closely, the validity of merging (ignoring register conflicts) is based on whether or not the current instruction differs from the previous instruction in the third bit of the P field. All of these tests serve to quickly reduce the number of potential merges. They are also inexpensive in time since they will be overlapped with other operations. Whether or not to check for register conflicts is a decision which should be made.

It is possible to avoid checking for register conflicts by assuming that the compiler would not generate a pair of instructions which can be merged, if there was register conflicts. This implies that the compiler would either have to re-order the instructions it generates, or insert some form of null operation to assure that the invalid merging would not be possible. Taking this route shifts responsibility to the shoulders of the compiler writers.

One problem with assuming that the compiler writers are responsible for invalid merge avoidance, is that there are far more compiler writers than hardware designers. If the hardware assures invalid merges do not happen, only one person has to “do it right”. Checking for register conflicts in hardware can be overlapped, to a great extent, with time taken for other operations. If the previous instruction's Y field matches the new instruction's Y or Z field, there is a potential register conflict. This is only a potential conflict as a STORE instruction can proceed at the same time as an ADD with the same register, provided the STORE came first. Making this distinction would take extra hardware, and possibly extra time. Skilful compiler writers can delay stores until necessary. Consider the two source level statements:

\[
x = a + z; \quad y = b + a + x*2;
\]

and note that while this would result in eight instructions with or without delayed storing, delayed storing would allow the merging of instructions to reduce it down to six effective instructions. It is possible to allow compiler writers to make the situation better, but not worse.

Given that the hardware has checked that the merge is valid, all that remains is to bitwise or the new instruction with the previous instruction.

Whether or not a merge was possible, the new instruction now has to be given to the cache section to keep. The ID presents the new instruction, with the instruction pointer of the first packet in the instruction, to the cache section and raises a signal asking for it to be picked up. This leads to the consideration of the cache section next.
The cache section holds a number of instruction and address entries. They are ordered by the loading sequence. Associated with each entry is a one-bit field termed \textbf{USED}. When a new instruction is accepted from the \textbf{ID}, this bit is set to zero for the new instruction. In order to provide space for the new instruction, all entries in the cache are moved "forward" one position. For long bursts of sequential code the cache acts as a \textbf{FIFO}. A new entry from the \textbf{ID} cannot be accepted if the \textbf{USED} bit of the entry at the head indicates that the entry has not yet been requested by the \textbf{FC}. The \textbf{ID} can continue in a mindless fetching and decoding of sequential instructions without any consideration of how long it is going to take to execute any of the instructions in the sequence.

This scheme is complicated by the existence of merged instructions. If two instructions have been merged, the second, even though stored in the cache, will not be requested. The setting of the \textbf{USED} bit has to be applied to both the entry requested, and the next entry if the requested entry is merged. The \textbf{FC} requests an entry for one of two reasons. It either is going to pass the entry to the \textbf{CLU}, or is going to discard it because the previous \textbf{IF} instruction indicated that an instruction is to be ignored. The cache has to be made aware of this distinctions. The \textbf{USED} bit of the next entry can only be set if the requested entry is merged, and the requested entry is not to be ignored.

These complications aside, the cache section is simple. It sits quietly until either the \textbf{FC} requests an entry, or the \textbf{ID} attempts to give it one. If the \textbf{ID} presents an entry and the \textbf{USED} bit of the first entry is set, the entries are moved one forward, the new entry accepted, and the \textbf{ID} turned loose to obtain another instruction. If the \textbf{USED} bit of the first entry is not set, the \textbf{ID} is ignored. Ignoring the \textbf{ID} is acceptable, as the \textbf{FC} will eventually request the first entry, and the new entry from the \textbf{ID} can be taken.

If the \textbf{FC} has requested, the cache matches the address of the entry with the address of the request and either finds it or not. If the entry is found, the \textbf{USED} bit or bits are set, and the entry given to the \textbf{FC}. If the entry is not found, one of two paths have to be taken. If the \textbf{ID} is actively decoding a sequence, it can safely be assumed that the next instruction from the \textbf{ID} will be the one requested. The new entry from the \textbf{ID} can be passed to the \textbf{FC} when it arrives. Because the \textbf{ID} cannot safely continue loading and decoding if it does not "know" what instruction is next, it will be actively chasing a sequence only when it is truly a sequence.
If the ID is not active, the cache has to request an instruction by giving the ID the address requested. Since the ID is waiting for this, everything can merge with the path taken when the ID was active, since it now is. All that needs to be worried about is that the cache knows when the ID is active. This is easily assured by having the ID provide an indication with each new entry. The active signal is best described by the phrase, “and I'll be back,” which is appended to the normal messages from the ID to the cache. Considered as a limited script for a set of actors the state transitions which the cache must follow is obvious if the “script” it has been given to follow indicates that the other two “actors” are limited to the lines:

FC: Could I have location X, and I need it.
FC: Could I have location X, as a matter of principle.
and
ID: Here is location X, and I'll be back.
ID: Here is location X, and I'll wait.

The phrase terminating the lines of the other two players is sufficient for the cache to maintain a consistent view of the full environment.

Turning to the other side of the ID, the PACKET LOADER (PL) has to be addressed. The PL is best described with reference to figure 5.48. The simplest part is obviously the ROUTER which passes the upper or lower 16 bits of the word depending on the value of the E bit. Provided the instructions are accessed sequentially, each word loaded supplies two instruction packets. A sequential accessing of packets is the most convenient way to start looking at the PL is detail.

Whenever the NEXT PLEASE signal arrives, the INCREMENTER OR PASS THROUGH (ILOPT) either accesses a new word, then has the ROUTER supply the first packet, or has the ROUTER supply the second packet from a data word. When the second packet has been supplied, this signals the ILOPT that the address it contains has to be incremented. This incrementing can be done while the ID section of the IFU is handling the packet just provided. To assure that the incremented value is available as soon as possible, the value can be incremented as soon as it has been passed out to obtain a new data word. This time will overlap with the memory access time. There is no special need for an extremely fast adder to perform the increment. A ripple carry circuit is sufficient. The ID section keeps providing the NEXT PLEASE signal, and the PL keeps supplying the instruction packets.
As well as providing the NEXT PLEASE signal, the ID section supplies what it perceives to be the address. With sequential access this will always agree with what the IOPT considers to be the address. Non-sequential access is defined as a discrepancy between these two addresses. In such a case, a memory access is required and the IOPT uses the supplied address rather than its pre-computed, incremented value. As well as using the address for a memory reference, it also goes into the increment part to prepare the next memory address. Here a ripple carry circuit can be a performance bottle-neck. If the address from the ID section is odd, the next packet will come from the incremented memory address, halving the time available for the increment part to obtain the new address. The memory would have to be very fast for any effect to appear.

The part of the IFU which is left is the FLOW CONTROLLER. Were it not for the fact that it handles most of the instruction pointer modification it would be reasonably trivial. For those instructions not handled locally it is. The FC asks for an instruction from the cache (through the SILS), and, if it is not for local processing, it waits until the CLU asks for the next instruction by sending a modification message for R12. It then sends out a modification message for R11 if needed, followed by a modification message for R10. Concurrently, it uses the value of the Q field of the instruction to modify its local version of the instruction pointer, R9. This cycle gets repeated for all the simple instructions in a sequence. Figure 5.49 gives the state transitions and actions that the FC follows. Further discussion will be with reference to this figure.
Figure 5.49 FLOW CONTROL State Transitions and Actions

The first thing which needs clarification about figure 5.49 is the meaning of the inputs. The inputs fall into two distinct classes. The smallest is the register modification class. As this is relatively simple it can be covered first.

An input of \textbf{R9} means that some instruction has produced a new value which is to be stored in the instruction pointer, which will come about if a \textbf{FLYING LEAP} instruction, or the “fake” instruction from the \textbf{SILS} for loading \textbf{R9}, is executed. An input of \textbf{R8} comes from either the operation of the \textbf{AIU}, in which case it is producing a new carry value, or the “fake” instruction from the \textbf{SILS} for loading \textbf{R8}. An input of \textbf{R15} is the means used by the \textbf{AIU} to indicate whether or not the instruction following the \textbf{IF} instruction is to be executed. \textbf{R15} is not maintained by any unit in the processor, and was not listed in figure 5.37. The value of the least significant bit is used to determine if the next instruction is executed.
The other seven inputs are labeled with the values of bits extracted from appropriate positions within the internal instruction representation. These are (numbering from high to low starting with 0) \( P_1, P_3, S_0, S_2, \) and \( T_0 \). These five bits are sufficient information for the FC to correctly handle all instructions (\( Q_1 \) will be mentioned). A short definition of the meaning of each input is:

- **00000** - An instruction which is simple
- **10000** - An IF instruction
- **01000** - A SWITCH instruction found in the program
- **01110** - A SILS SWITCH instruction to manipulate the FREE/LOAD lines
- **00110** - A CALL instruction
- **00100** - A HOP or JUMP instruction, \( Q_1 \) determines which
- **00001** - A FLYING LEAP or SILS instruction changing \( R_8 \) or \( R_9 \)

With respect to the listed actions, GIVE means that one or two messages will be sent out the communications ring. These will be addressed to the CLU, and will update \( R_{10} \) and possibly \( R_{11} \) if necessary. The actions ASK NEED or ASK PRINCIPLE obtain another instruction from the cache, with the appropriate indication so that the cache can correctly handle the USED bits. When obtained, the value of the Q field of the new instruction is used to update the value of \( R_9 \). Action \( m \) needs a few words. Register \( R_8 \) will be updated by the AIU and loaded by the DAU. Action \( k \) is implicit in the loading by the DAU, leaving action \( m \) for the AIU. The bit affected by the AIU is the bit used to store the carry, which is the most significant bit of \( R_8 \). Action \( m \) replaces the upper bit of \( R_8 \) with the upper bit of the value received from the communications ring.

The state transitions in figure 5.49 are slightly simplified. What does not appear there, is any indication of how requests from the CLU are handled. It is assumed that a reception of \( R_{12} \) sets a flag. The GIVE action will stall until the flag has been set, then will clear it as part of its task. Attempting to integrate \( R_{12} \) into the transition table would overly complicate it, and would not reflect how the implementation works.

The effect of the FIFO FULL input has to be mentioned. It has been left out of the transition table completely. What effectively happens is that when an ASK NEED action is attempted, if the FIFO FULL input is present, a SWITCH instruction with a Q field of 000 is simulated.
Two of the five units, the CLU and the IFU have now been covered. There is little of interest within the MAU and it need not be discussed. The AIU and DAU are also rather conventional except for a few details.

**Arithmetic and If Unit and Data Access Unit**

Both the AIU and DAU hold copies of the eight general purpose registers. Any consistency that matters is assured by appropriate messages on the communications ring. These eight registers are stored on a set of rotating shift registers. This is a simplification of the register storage concept which was covered in the thesis of R. Atkinson (1988).

The scheme used here can be a simplified version of that presented in the thesis, because there is only one active element on the storage ring. There is no need for tags or other extra information to be carried with the registers, as the one active element can "know" which register is currently passing it. The storage ring can cycle at maximum speed since there is no FMIFO queue for storing partial instructions, as was presented in the thesis. All of the space benefits of such a scheme are available, with few of the costs. The space benefit is what makes keeping more than one copy of the registers a viable proposition. Two copies of the registers, stored as rotating shift registers, occupies much less space than a single conventional copy of the registers.

The asynchronous interaction of the units connected to the communication ring allows the AIU to implement its adder circuit in any one of many various ways. A ripple carry adder, while slow, takes little space. A parallel adder, while large, is fast. A predictive carry adder falls between these two extremes, both in size and speed. Which is the most appropriate is difficult to determine without extensive gate level simulation of large amounts of, what would be typical software, generated by a reasonably cunning compiler. A predictive carry adder would seem the best choice for a first attempt. The time taken for a ripple carry or parallel adder is fixed by the circuitry. The time taken by a predictive carry adder is dependent on the values being added. Results from simulation of such a predictive adder would allow statistics to be gathered on how much of a bottle-neck both a ripple carry and predictive carry adder would be in comparison to a parallel adder.

The interaction of the AIU with the other units on the communication ring is quite limited. When a modified register (R0 through R7) is received, the value is updated in the storage ring. When a modified R8 is received, the upper bit is extracted and used as the
value of the carry bit. When an instruction message arrives, the appropriate operation is done. At the termination of the operation the modified register (if there is one) is sent out on the ring, addressed to all units but the AIU. In the case of an IF instruction, a (IFU)(R15)(x) message is sent to provide the IFU with the information needed to decide if the next instruction is to be executed. Arithmetic or logical operations send a (IFU)(R8)(x) message to have the IFU's version of the carry flag correctly set, should the value of the carry bit have changed.

The DAU is slightly different. It maintains registers R0 through R7 and will update the stored values as they are received. Reception of both R12 and R13 causes it to start operation. When the operation is started a (CLU)(R13)(x) message is sent to inform the CLU that both messages have been received, and that the CLU can proceed to place another request for the DAU on the communication ring. On completion, should the operation have modified a register, that register will be sent out on the ring addressed to all units but the DAU.

Summary

The processor is a large collection of communicating hardware processes, passing messages between themselves in an asynchronous manner. Each part has its rigidly defined area of expertise, and a rigidly defined means of communication. The analogy to processes which exist within the operating system of the machine is strong.

The communications ring provides a means of allowing separate units to function in a manner independent of the internals of all other units. Each unit will take as long as necessary to perform the task it has to do. Over time, as units are recognized as bottle-necks and re-implemented to alleviate these shortcomings, the performance of the processor will improve, with no need to consider implications of changes with respect to the other units.

This is not a pipelined processor. Internal to the IFU there is some amount of what could be termed pipelining, in that instructions are being fetched and decoded and shuffled in the cache and prepared for the communication ring all at once. Folding ADD and STORE instructions departs slightly from the pipelining model, in that a pipeline has the AIU or DAU step first. Here the pipeline splits into two parallel pipes, to allow either the AIU or the DAU to come "first". At the CLU the pipeline model breaks down completely and is replaced by something akin to a parallel processing situation. Within the IFU, the
execution of the instruction pointer modification instructions, without recourse to the CLU, short circuits some of the “pipeline”.

By its very nature as message passing based hardware, there is no cyclic churning of instructions being executed. Instead, a web of flowing activity vibrates with the task at hand. The methodical plodding of a pipelined architecture is replaced with the ephemeral interactions of small components, each contributing its part to the total operation.

The external appearance of each part has been covered here, each with a plausible internal implementation. Just as the adder in the AIU can be implemented as a ripple-carry, predictive-carry, of full parallel adder, the internals of each part can be modified as needed to produce a total processor of varying power.

5.2.7 A CO-PROCESSOR Unit

A co-processor has to integrate smoothly with the rest of the machine. Three major aspects are:

1/ how does it detect that it has been addressed
2/ how does it obtain its operands
3/ how does it return its results

Implementing a co-processor as a replacement for a set of subroutines is trivial, provided the processor just covered is the processor in use.

Detecting that it has been addressed is trivial. Consider a co-processor for floating point operations. There is some set of operations it performs. First the subroutines that the co-processor replaces need to be looked at.

These subroutines exist in some section of the address space. Each new improved release of the subroutine library will have these stored at possibly different addresses than the last release did. They may even be of different sizes if they where involved in the improvements. In order to cater for these changes, without requiring the re-compilation of programs which use these subroutines, they have to be indirectly addressed in some manner. Various means of indirect addressing exist, but all essentially reduce to a table of addresses which are at some fixed location. The indirect means being considered is a table of JUMP instructions. JUMP instructions are three units long so each will be separated from the next by a one unit null operation, making each JUMP instruction start at an address which is a multiple of four. Make the size of the table a power of two by padding it
with an appropriate number of null operations. Assure that the table starts at an address which is a multiple of this power of two.

The address of this jump table (ignoring the bits which index into the table) is the address that the co-processor recognizes. Which operation to perform is indicated by those bits which were ignored. The co-processor monitors the address lines from the processor. When the address matches it “knows” that it has been addressed, and what it is to do.

![Diagram showing CO-PROCESSOR Selection Detection](image)

**Figure 5.50 CO-PROCESSOR Selection Detection**

Figure 5.50 shows how the co-processor detects that it has been selected. When the co-processor is not selected it introduces one gate delay into the propagation of the ADDRESS VALID signal. It will be noticed that selection involves a valid address, and that the location is selected for reading and not writing. The address space which contains the jump table can be initialized at any time. Normally the area containing the library is ROM, but can be implemented with RAM, initialized at startup by loading from some medium. The restriction to read-only accesses allows such a solution.

The co-processor “knows” that it has been selected, and it “knows” what it is supposed to do. It now needs the operands to work with. This has to be discussed in relation to the subroutines which it replaces.
A subroutine must "know" where to find its operands. Assume that the method chosen by the compiler is known to the persons designing the co-processor. For convenience assume that this method passes the first four words worth of arguments in registers R1 through R4. All other arguments are passed on an execution time stack.

The co-processor must have some means of obtaining the values that the processor has in registers. The processor does not normally make the contents of its registers available outside of itself, except in one special situation. If the process currently running relinquishes the processor, or is forced to do so, the contents of the registers are saved to memory. It should now be clear how a co-processor obtains the values of the registers.

When the co-processor detects that it is selected, it provides, as the contents of the addressed memory location, a zero. The processor interprets this as a SWITCH instruction. It saves the contents of all the registers to locations $00000000$ through $00000013$ using ten writes to memory. It then pulses the FREE line, and waits for a pulse on the FIFO FULL line. The co-processor, after generating the "SWITCH" instruction, can collect the values of the registers as they are "stored". These values are not passed through to the memory since they are only needed within the co-processor.

The co-processor now has all the information it needs to perform its task. For the hypothetical floating point co-processor, for example, a single precision multiply may multiply the contents of R1 by the contents of R2, returning the result in R1. For quadruple precision R1 and R2 may be pointers to areas in memory where the operands are found, with R3 pointing to where the result is to be stored. The processor is now in an idle state, and will not do anything until the FIFO FULL pulse arrives, allowing the co-processor to act with all the capabilities of a processor, such as reading and writing memory locations.

After completing its task, and updating its copy of the register contents as necessary, the co-processor can generate the FIFO FULL pulse to have the processor "pick up" the result. The final operation left to do is return from the "subroutine" that seemed to consist of only a SWITCH instruction. It is far too complex to attempt to supply an instruction to return. Since the co-processor has access to the contents of the registers, and one of these holds the return address and another holds the instruction pointer, it is far easier to provide the contents of the return address register as the content of the instruction pointer register.
To the processor, it called a subroutine at some location, picked up the first instruction which was a SWITCH instruction, stored the process away, was told to pick up a new process, and did so. It does not matter to the processor that the process is exactly the same one it put away, or that the instruction pointer picked up is not the same as that put down.

It should now be clear why, in figure 5.12, the co-processor has the FIFO FULL line running through it, and why the FREE and LOAD lines are covered. The co-processor has to generate a pulse on the FIFO FULL line, and must stop pulses on the FREE and LOAD lines from going to the other components on the processor board.

All of these operations within the co-processor are independent of the specific purpose of the co-processor. A "null" co-processor can be designed with all interfacing components already in place, and a large area of the chip left for the task specific part of the co-processor. The task specific processor is presented with eight data registers, and one command register (the WHAT from figure 5.50) which tells it what it is to do. Provision for a memory address register and memory data register can also be made. It performs its task, updates the data registers, and signals completion to the independent section of the chip, which can then let the processor continue.

This scheme is acceptable in a multiple processor machine, where it would not be in a single processor machine. Should a co-processor operation take a very long time to complete, there is no way of interrupting it. For a single processor machine such long disable times are not acceptable, but if one of the many processors is busy for a period it is not as important, as other processors are available.

There is one area of concern which has yet to be covered. If the last word necessary to force the generation of the FIFO FULL pulse arrives at the ROUTE & COUNT component on the processor board, just as the processor is attempting to load the JUMP instruction, the co-processor must not provide the SWITCH instruction. If it did so, the FIFO FULL signal from the ROUTE & COUNT component would be “lost”, as it would have passed the co-processor. This is overcome by having the co-processor monitor the signals on the LOAD and FIFO FULL lines and maintain itself in one of two states.

Initially the co-processor is in an “active” state. When a FIFO FULL pulse passes it goes into a “passive” state. When a LOAD pulse passes it goes into the “active” state. The co-processor will only generate a SWITCH instruction if it is in an “active” state. There are
two alternatives for what to do in the "passive" state. The co-processor can let the JUMP instruction be read, but it does complicate the circuitry seen in figure 5.50. Detecting that it was selected has to depend on the state. The JUMP instruction, being three units long, implies that two reads are needed to obtain the full instruction. A side effect of this is that the process will not use a co-processor for its task when it is again given a processor, even if one is available with the processor to which it is re-assigned. This co-processor will have assured that the instruction pointer is addressing the subroutine and not the JUMP instruction.

A second alternative does not complicate the detection of selection, and does allow the process to use a co-processor of the processor to which it is re-assigned, if a co-processor exists. When in the "active" state, the co-processor provides a SWITCH instruction. When in the "passive" state the co-processor provides a self referential HOP instruction, but does not go into operation. The processor will then detect the FIFO FULL pulse, save the state, load a new state, and continue. The saved process, when re-assigned will start by attempting to load an instruction from the same location and, if a co-processor exists with that processor, the co-processor will be used.

While performing its task, should the co-processor detect that a FIFO FULL pulse arrives it can hold it until complete, and after generating its own FIFO FULL pulse and returning the register contents, pass the FIFO FULL pulse along. All race conditions, and timing considerations can be avoided.

Any number of co-processors can be installed between the processor and cache on the processor board. No more than one can be addressed at any point in time. For a basic machine no co-processors need exist. Replacing a co-processor with a plug, which routes signals straight from one side of the socket to the other, is all that is needed to maintain correct operation of the full processor board. Should an existing co-processor become faulty, it can be pried out, and a blank plug used to replace it. The switch settings which are used to inform the kernel of which co-processors are where should be made consistent with the new state, but failing to do so would only mean that the kernel may not choose processors for processes as well as it might.

The integration of co-processors into the whole processor board is as "seamless" as possible. The convenience of having one executable image for a program, whether co-processors exist or not, is obtained. The processor does not have to be designed with the knowledge of the existence of co-processors, so no extra "instructions" have to be designed.
into it. When a co-processor is detected as faulty, it can be removed. Should the switches informing the kernel of its presence be incorrectly set, the only effect is that the kernel will make less than optimal choices of which processor to assign certain processes to. Correctness is guarantied, only performance will suffer from incorrect switch settings.

The discussion of the general processor board is complete, and the processor board used by the kernel of the operating system can now be covered. This board is slightly different from a general processor board.

**5.2.8 Kernel Processor Board**

The kernel processor board is shown in figure 5.51. Flicking between figure 5.12 and figure 5.51, it can be seen that they are remarkably similar. Some of the wires not shown on the diagram of the kernel processor may still be there. The only major breaks are that the **FREE** line no longer goes to the co-processors, and the **LOAD** line does not go to the **CACHE**. All wires not shown in figure 5.51, that were on 5.12, are inactive because the changes which would cause them to be useful never happen. The major change is that the **ROUTE & COUNT** part of 5.12 has been replaced by the **FIFO** and **WORK TO DO** parts.

All components on the kernel processor board which are the same as on the general processor board are exactly the same. Any co-processors still perform as they did before, the **CACHE** and **MMU** still cache and map addresses. Because the **CACHE** never receives a **LOAD** signal, it never identifies which segments are to be discarded. Because the **MMU** never receives a **LOAD** signal, it never leaves its initial state which does one-for-one mapping of addresses, with all possible offsets valid. The **MASTER** never receives either **FREE** or **LOAD**, and so never attempts to write the processor descriptor word to $C000XXXX as the **MASTER** units on the other processors do.

The **DEVICE** accepts addresses of the form $C000XXXX where XXXX can be any value, as this is the kernel processor. The other processor board **DEVICE** units accept $C001YYYY where YYYY is the number of each specific processor board.

As with all **PROCESSOR** units on all processor boards, the kernel **PROCESSOR** starts by generating a signal on the **FREE** line, then waits for the **FIFO FULL** line to go high. The other processor boards will receive a **FIFO FULL** signal when the kernel has given them a process to execute. The kernel processor **FIFO FULL** signal can be
generated when the first general processor has written its description to the kernel processors DEVICE. At initialization, the FIFO FULL signal is automatically generated. This gets the discussion to the WORK TO DO unit.

![Diagram of the Kernel Processor Board](image)

**Figure 5.51 The Kernel Processor Board**

Normally the WORK TO DO unit generates a FIFO FULL signal after receiving both a FREE and a FIFO NOT EMPTY signal. The FIFO NOT EMPTY signal is maintained by the FIFO as long as the FIFO is not empty. When the kernel has completed all work that it knows it has to do, the kernel program executes a SWITCH instruction. This generates a FREE signal which, if there is data in the FIFO, causes an immediate FIFO FULL signal. Should the FIFO be empty this signal is delayed until some data has
been written to the FIFO. The WORK TO DO unit assures that when there is nothing for the kernel processor to do, it does not use any bus bandwidth.

The DEVICE unit was discussed earlier as if it was a write-only device. The DEVICE unit supports both reading and writing. For general processor boards no reads addressing the DEVICE are ever attempted. When the kernel processor continues after its SWITCH instruction there is at least two words in the FIFO for it to read. When it does read it obtains the address of a processor, and the description of that processor. What the kernel does with this information has been covered in an earlier chapter.

Properly built, the FIFO and WORK TO DO units can be included in one component. That component can be pin compatible with the ROUTE & COUNT unit. Changing a general processor board to a kernel processor board consists of removing the ROUTE & COUNT unit, replacing it with the kernel specific component, disconnecting one strap which carries the LOAD signal to the CACHE, and changing the switches which determine the address that the DEVICE will respond to. While the kernel processor is a critical component, without which the whole machine is useless, the kernel processor is defined by one critical component. Should any other component of the current kernel processor board become faulty, a new kernel processor board can be "installed" by a few modifications to any other general purpose processor board.

Because the CACHE is a pending-write cache, it is quite possible that when the kernel executes a SWITCH instruction, and the contents of the registers are saved to memory, the changes to these "locations" may never make it out of the CACHE. To an outside observer it will appear as if the kernel "knew" when there was information in the FIFO, and when there was not. The kernel can present a minimal load on the bus. Given a sufficiently large cache, which when the kernel is written properly is not that large, there will be no need for the CACHE to ever perform any of its pending writes, nor replace any stored values due to lack of space. The original kernel of the PORT operating system was less than 4000 bytes of Intel 8086 code, and used a little more than 2000 bytes of data storage, beyond the 64 bytes for each process descriptor. The parts of the PORT kernel which could have been cached would have been completely held, with room to spare, in 8192 bytes of cache. After initially loading the code for any functions it needs, the only burden the kernel would place on the bus would be to read the information from its FIFO, and to update the values stored in shared memory that other processes need access to. All
private information would remain totally private, stored in the CACHE of the kernel processor board, accessible only from the kernel.

Another benefit of total caching appears when initialization is considered. All other processors are given a valid memory mapping that guarantees that the instruction pointer loaded after a SWITCH instruction is reasonable. The kernel can arrange that the value stored in the appropriate location is what it should be. There is no active process to do this on the kernel’s behalf. The address space divisions shown in figure 5.10 are not quite as they appeared. The first small part of what was shown as RAM is ROM, and is large enough to contain the initial values of the kernel’s registers. These locations are never accessed after initialization, because all future “references” to these locations will be satisfied by the CACHE on the kernel processor board. The register contents stored in this ROM provide the initial values of the kernel’s registers, and any changed values stored by the SWITCH instruction never make it past the CACHE.

With respect to devices rather than other processors, the devices also write to the kernel processors DEVICE unit to announce their status. If the definition of a processor number is that it is less than 32,768, and that a device number is between 32,768 and 65,534 inclusive, the kernel can also receive device status information in exactly the same way as it receives processor status information. To the kernel, a device is a processor which handles its own dispatching. As was seen in chapter 4, the Wait_For_Event request is the only request that the kernel provides for handling devices, and all the kernel does is suspend a process using that request, until a status from the appropriate device is found in the FIFO.

With respect to devices, how they are integrated into the machine is worth a comment here. There needs to be some way for the kernel to be made aware of the existence of the devices. There also needs to be some means of causing a process to come into existence to interface for that device. Each device is assigned to a 131,072 unit address space. The first half is the device interface itself. The second half is a piece of memory which contains the program for a process to interface for that device.

When the kernel starts executing, it is the only “process” in existence. It reads the first word of the memory for each “device”. If the word is non-zero, the device exists, and the kernel “knows” what process to create to interface for that device, and that that process is to be given the ability to address the device interface area. A board with a WIZZ-BANG Z/46 can be plugged into any machine, and that machine will automatically have the device
installed, with a process to support it, when the power is reapplied. This is not a feature which is unique to this machine, but it is not common. The initial device process can create other processes as needed to provide complete support for the device.

For “non-stop” computation, it is even possible to build boards which can survive transients on insertion and removal. If the kernel is given an extra primitive which tells it to check for new devices, installation can be done without interrupting any other task. Insert the new board, tell the kernel to go and look, and the interfacing processes will be automatically created. The removal of a board will be detected when the process interfacing to it has to read an instruction from its device specific memory. This will result in a SWITCH instruction which informs the kernel that a “bad” memory location was accessed, and the interfacing process will be destroyed. Properly structured, another process can inform a human that the WIZZ-BANG Z/46 has ceased to be usable on the machine in question.

Both general and kernel processor boards have now been covered. Discussion can turn to some of the more interesting devices which make the machine useful.

Section 5.3 The User Interface Device

To be used by a person, a machine must have some means to support input from a person, and some means of presenting output to that person. The area of the user interface will be an open research area for the foreseeable future, involving persons from many diverse fields, and will not be addressed here. A reasonably high resolution bitmap display with a keyboard and some form of pointing device is envisioned. Interest has been focused on certain aspects of the display technology, and on the integration of the user interface device with the rest of the machine. Figure 5.52 shows the overall view of the user interface device.

Most noticeably, the device appears externally as two pieces. Both input and output, while interrelated, are two distinct tasks. As a quick overview, the INPUT INTERFACE MEMORY (IIM) and OUTPUT INTERFACE MEMORY (IOM) are two dual ported pieces of memory, plus control locations, through which all communications with the device take place. The BUS MASTER is used to write to the FIFO of the kernel processor board. The PROCESSING ELEMENT (PE) consists of one of the processor chips previously described, ROM containing the program that the processor executes, and RAM for use as a data area. The DISPLAY MEMORY (DM) is a piece of
specially organized memory which contains the bitmap which will be displayed. The **KEY AND POINTER** input area is a socket area. Various sub-modules can be plugged in to support various types of keyboard and pointer. The other two major components are typical implementations of their respective types and are not worthy of further discussion.

Figure 5.52 The User Interface Board

### 5.3.1 Input Interface Memory

The **IIM** consists of two separate sections of memory, an input buffer and a control location. The control location is written by the interface process which represents the device within the operating system, and read by the **PE**. The buffer is written by the **PE** and read
by the interface process. The buffer is some convenient size, determined by the availability of memory chips. At least 512 16-bit words will exist. The buffer carries input information for eventual use by other processes, while the control location is used to inform the P.E. of the status of the input buffer. The buffer is addressable by either the P.E or through the INPUT DEVICE by the interface process, but not by both at the same time. The control location can only be read by the P.E, and only written through the INPUT DEVICE.

The control location does not exist, there is only the buffer. The use made of the buffer by the P.E is as the destination of a store operation. The use made of the buffer by the interface process is as the source of a load operation. If the P.E attempts to read from the buffer, this is interpreted as reading the control location. If the interface process attempts to write to the buffer, this is interpreted as writing to the control location. The control location is implemented as a single stored bit in the memory access circuitry. The bit contains a one if the last operation which addressed the buffer "incorrectly" was a write by the interface process, and a zero if the P.E was the last to access it "incorrectly". The bit forms the least significant bit of the value "read" by the P.E.

The choice of the least significant bit to carry a control signal was made based on the knowledge that the instruction set of the processor makes testing this bit inexpensive. Initially the P.E can assume that the interface process is busy, and will not provide any input until told to do so. No event will be generated until the interface process first requests it. The mere existence of this interface process assures that the kernel has all its data structures in place. The buffer is initially only addressable by the P.E.

When the P.E reads a control word which requests input it will copy, into the buffer, input which has arrived since the last time it dispensed input. It can make the buffer addressable through the INPUT DEVICE, and write a word to the kernel to announce that an event has occurred. Should no input be available it can remember that input has been requested, and can pass it through when it does arrive.

Having the buffer be addressable by only one user at a time means that there is no need to use dual ported memory. The P.E will only access the buffer after it has read a control word requesting input, and before it writes the word which the kernel will interpret as signalling an event. The interface process will only access the buffer after it has detected an input event, and before it requests more input.
The pointing device is assumed to provide a means of specifying a point within a volume of three-space which can be described by a triplet of numbers which have values between 0 and 16,383. The keyboard is assumed to have no more than 8,191 keys. Each entry in the input buffer will contain an indication of what the other 14 bits contain, by a specific value in the least significant two bits. A value of 00 indicates that a key is the source of the input. A value of 01 indicates that the pointer has moved to a new value in the X direction. A value of 10 indicates a new Y value for the pointer, and a 11 indicates a new Z value. For a two dimensional pointing device the Z value will never change, the X coordinate corresponds to the left/right movement and the Y coordinate corresponds to a top/bottom movement. A motion sensing pointing device such as a mouse reports changes rather than absolute values. To support this type of a device, the pointer is assumed to be initially at a coordinate of (0,0,0). A position sensing device such as a tablet provides absolute values and any assumed initial position is corrected on the first value read.

The PE will assure that any pointer changes will be completely reported within one input buffer. If the pointer only moves horizontally, only a X pointer input will be supplied. If it moves in two coordinates, two input values will be required, and both coordinate values will be passed in the same input buffer, adjacent to each other. The interface process can assume that it is able to work with complete information on a buffer by buffer basis.

The keyboard is assumed to be capable of generating key codes and an indication of the change in the key in question. If the key is depressed the most significant bit of the unit will be a one. When released the most significant bit has a value of zero. The key code occupies the other 13 bits. Buttons associated with the pointing device have key codes assigned from $1FFF and descending. Figure 5.53 shows how each unit of input can be interpreted.
A buffer of input has the first unit containing a count of the number of units which contain input information within the buffer. Each unit contains either an indication of where the pointer has moved to, or contains an indication of a key press.

5.3.2 Output Interface Memory

The OIM is much like the IMM in hardware. The major difference is that the write and read aspects have changed ownership. The PE reads the buffer while the interface process writes it. This has a great effect on the implementation of the control location.

While the IIM can use “incorrect” access to the buffer to generate the control information, this is not possible with the OIM. The interface process still has to “write” the control location but it has to write the buffer as well. This is easily overcome because the information content of the “write” operation is embodied in the access mode and not the value involved. This half is easily solved by having the interface process “read” the control information. When the interface process reads the buffer this is sufficient to indicate that the control bit should be set.

Having the interface process “read” the control bit to a set value is easy, although contorted. It is far harder to have the PE “write” the value to determine what it is. Stepping back a bit, the buffer contains information describing the output, and a count describing how much output is present. If this count is the last piece of information that the interface process places in the buffer, addressing this location is sufficient to indicate that a new buffer is complete. When the PE reads this location there will either be a new value there or not. This is indicated by the value of the control location. If no new value is present, the PE does not get to access the location but rather is supplied with a value of zero. If the location has been written since it was last “read” by the PE, the true value is supplied.

Getting information into and out of the board has now been covered. Attention can turn to the interesting part, the organization of the display memory.

5.3.3 Display Memory

There are many ways to organize the display memory. Each graphical application has an organization which is “best”. Some applications are suited to an organization which allows all planes of each pixel to be changed at once. Drawing a circle is one example. Other applications are more heavily oriented to horizontal or vertical lines. Such tasks are made
more efficient by an organization which allows multiple pixels to be modified in one operation.

Four organizations will be looked at. In the PIXEL organization, one bit from each plane is grouped with the corresponding bits from the other planes. The other three organizations all keep the planes separate, but allow them to be accessed in parallel. The second organization, the HORIZONTAL organization, allows some number of pixels to be accessed in one operation. These pixels are adjacent horizontally. This is a common method found in many commercial products. The third, the VERTICAL organization, is like the HORIZONTAL, except that the adjacent pixels are vertical. While apparently equivalent, apart from a rotation, it can perform differently due to the relative aspect ratios of the operations which are required. The fourth organization, the COMBINED organization, assumes that both horizontal and vertical operations are possible.

The PIXEL organization is preferred if the display memory is to be addressed by applications in the same manner as any other memory. It is quite common in many personal machines found in current use. Single plane displays often group many pixels into one addressable cell, providing multiple pixel operations which approach the HORIZONTAL organization. The major advantages of a PIXEL organization are simplicity, and the ability to treat any segment of memory as the display memory. The major disadvantages of a PIXEL organization are directly related to the advantages. The simplicity which is beneficial for many applications, becomes a complication for others. Consider a graphical editor for VLSI layout. Treating each plane as a different layer simplifies much of the work within the editor, but the PIXEL organization implies that to clear a level each stored pixel has to be picked up, masked down to remove the bit representing the level, then placed back into memory. Inherent in this description was another problem. For displays with a large number of planes, each pixel has to be accessed separately. This can imply a very large number of accesses for a large area. The problem with allowing any segment of memory to be used as the display memory is that refreshing the display will require a goodly percentage of the bus bandwidth to access the display memory. While acceptable when a single processor relatively slow machine is envisioned, when multiple processor machines are considered, this constant load on the bus can become a bottle-neck. These disadvantages can far outweigh the advantages.

All operations on the display memory can be described as operations on a set of rectangular regions. Each region is H pixels wide, and V pixels high. A single pixel region
has \( H=V=1 \). A horizontal line has \( V=1 \) and \( H \) to the line length. A vertical line has \( H=1 \) and \( V \) set to the line length. The value \( H \) can be defined to be equal to \( 16*N + b \), and the value \( V \) to be equal to \( 16*M + a \). A PIXEL organization has to access \( H*V \), or 
\[
256*M*N+16*b*M+16*a*N+a*b
\]
locations to change the rectangular region.

A HORIZONTAL organization can support changing up to 16 pixels at once, with no requirement for the pixels to be aligned on native addressable boundaries. The display memory cannot be a segment of normal memory, but is special. Addressing this memory must be done in some non-standard way, which is a complication, but the ability to ignore alignment is a simplification. Single plane PIXEL organizations can access multiple pixels but the alignment requirements are a complication. Multiple plane displays reduce the number of pixels available to the PIXEL organization, but not the HORIZONTAL, because the planes can be accessed in parallel. Because a special memory is needed to support the HORIZONTAL organization, programs have already had to “admit” that they are accessing display memory, and so being “more different” from normal memory is acceptable. The benefits in access counts can be clearly seen. Again, for the \( H*V \) rectangular region which splits into the same \( 16*x \) parts, the HORIZONTAL organization requires \( 16*N*M+a*N+16*M+a \) accesses, which probably is over an order of magnitude less than the PIXEL organization. Given that accessing this display memory has to be performed in a special manner, it can be placed in a special place so that the refresh operations do not impact bus bandwidth, which is a considerable saving in a multiple processor organization.

The VERTICAL organization is the HORIZONTAL with a rotation. The same \( H*V \) rectangular region requires \( 16*N*M+b*M+16*N+b \) accesses. For the same 100 by 100 region the same number, 700, accesses is required as with the HORIZONTAL organization. The difference between these two becomes evident when the region is sufficiently non-square. A 100 unit long horizontal line requires 7 HORIZONTAL accesses but 100 VERTICAL accesses. The same line drawn vertically swaps these numbers.

The COMBINED organization assumes that both HORIZONTAL and VERTICAL operations are possible. How the memory is organized will be seen shortly. For any given rectangular region, either the HORIZONTAL or VERTICAL organization is better, or they are equivalent. It is swiftly evident that the COMBINED will perform as
well as the better of the two in all cases. While it may be clearly evident that this is the case, it is not true. Figure 5.54 shows how the H*V region is split into four sub-regions.

![Figure 5.54 Divisions of a Rectangular Region](image)

The upper left region is a multiple of 16 in both height and width. For this region both the HORIZONTAL and VERTICAL organizations require the same number of accesses. The upper right region is best served with a VERTICAL organization. The lower left is best served by a HORIZONTAL organization. Which is best for the lower right depends on which of \( a \) or \( b \) is smaller. The formula for the number of accesses with the COMBINED organization is \( 16^*N^*M+a^*N+b^*M+\min(a,b) \). What was "evident", is only true if one of \( a \) or \( b \) is zero. For the example 100 by 100 region \( N \) and \( M \) are both 6, and \( a \) and \( b \) are both 4. The COMBINED organization has to perform 628 accesses rather than 700. While not as dramatic a saving as that obtained when going from the PIXEL to HORIZONTAL or VERTICAL organizations, saving another 10 percent is acceptable.

![Table 5.55 Access Count 14 Point Times ASCII](table)

<table>
<thead>
<tr>
<th>HEIGHT</th>
<th>WIDTH</th>
<th>FREQUENCY</th>
<th>PIXELS</th>
<th>HORIZONTAL</th>
<th>VERTICAL</th>
<th>PROPOSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4</td>
<td>2</td>
<td>8</td>
<td>28</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>4</td>
<td>784</td>
<td>196</td>
<td>56</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>5</td>
<td>560</td>
<td>112</td>
<td>40</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>6</td>
<td>672</td>
<td>112</td>
<td>48</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>7</td>
<td>3,038</td>
<td>434</td>
<td>217</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>8</td>
<td>572</td>
<td>84</td>
<td>48</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>9</td>
<td>1,008</td>
<td>112</td>
<td>72</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>10</td>
<td>1,680</td>
<td>168</td>
<td>120</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>11</td>
<td>308</td>
<td>28</td>
<td>22</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>12</td>
<td>336</td>
<td>28</td>
<td>24</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>1 4</td>
<td>13</td>
<td>364</td>
<td>28</td>
<td>26</td>
<td>26</td>
<td></td>
</tr>
</tbody>
</table>

| TOTALS | 9,506 | 1,330 | 679 |
|        | 100%  | 14.0% | 7.1% |

Figure 5.55 Access Count 14 Point Times ASCII
A major use of the display is as a character oriented device. It is worth while looking at how each of the four organizations perform in the area of number of accesses. Figures 5.55, 5.56 and 5.57 show this performance for three different point sizes of a particular typeface. All characters in each size are the same height, but the widths differ. The 95 printable ASCII characters are grouped by their widths. The number of accesses is based on drawing the complete set of characters once.

For the small 14 point size the COMBINED is the VERTICAL. The sizes of the characters imply that all modifications are done to rectangles which fall completely in the lower right sub-region of figure 5.54, and for the heights and widths used, the VERTICAL is the best choice.

<table>
<thead>
<tr>
<th>HEIGHT</th>
<th>WIDTH</th>
<th>FREQUENCY</th>
<th>PIXELS</th>
<th>HORIZONTAL</th>
<th>VERTICAL</th>
<th>PROPOSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>3</td>
<td>1</td>
<td>54</td>
<td>18</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>1</td>
<td>72</td>
<td>18</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>7</td>
<td>630</td>
<td>126</td>
<td>70</td>
<td>49</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td>12</td>
<td>1,296</td>
<td>216</td>
<td>144</td>
<td>96</td>
</tr>
<tr>
<td>18</td>
<td>7</td>
<td>3</td>
<td>378</td>
<td>54</td>
<td>42</td>
<td>27</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td>8</td>
<td>1,152</td>
<td>144</td>
<td>128</td>
<td>80</td>
</tr>
<tr>
<td>18</td>
<td>9</td>
<td>29</td>
<td>4,698</td>
<td>522</td>
<td>522</td>
<td>319</td>
</tr>
<tr>
<td>18</td>
<td>10</td>
<td>8</td>
<td>1,440</td>
<td>144</td>
<td>160</td>
<td>96</td>
</tr>
<tr>
<td>18</td>
<td>11</td>
<td>4</td>
<td>792</td>
<td>72</td>
<td>88</td>
<td>52</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>3</td>
<td>648</td>
<td>54</td>
<td>72</td>
<td>42</td>
</tr>
<tr>
<td>18</td>
<td>13</td>
<td>13</td>
<td>3,042</td>
<td>234</td>
<td>228</td>
<td>195</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>2</td>
<td>504</td>
<td>36</td>
<td>56</td>
<td>32</td>
</tr>
<tr>
<td>18</td>
<td>15</td>
<td>1</td>
<td>270</td>
<td>18</td>
<td>30</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>16</td>
<td>1</td>
<td>288</td>
<td>18</td>
<td>32</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>2</td>
<td>612</td>
<td>72</td>
<td>68</td>
<td>40</td>
</tr>
</tbody>
</table>

| TOTALS | 15,776 | 1,746 | 1,764 | 1,074 |
| PERCENT | 100% | 11.1% | 11.2% | 6.8% |

Figure 5.56 Access Count 18 Point Times ASCII

The 18 point typeface shows the first indication of the holistic aspect of the COMBINED organization. The characters which are 18 high and 16 wide are the only ones to not benefit from the use of the ability to mix vertical and horizontal operations. Mixing allows a minimization of the number of wasted pixel accesses.

The 24 point typeface shows more instances of the improvements possible when each of the four sub-regions is done using the most appropriate mode.
There is one other organization which is possible, and it needs to be mentioned. It is possible to arrange the chips so that any four by four square can be accessed in one operation. For these font examples, the 14 point font uses 808 accesses for the square organization, which is 18.9% more than the proposed. For the 18 point font it uses 1360 which is 26.6% more. For the 24 point font it uses 1908 which is 3.6% less. The fortunate choice of a 24 point font fits perfectly with a square organization. Provided the display is used for fonts which are multiples of four in height and width the square organization is perfect. Any other font choice appears to lean to the proposed solution.

Drawing characters is an important aspect of the display's use, but graphics is also of reasonable import. Some examples of the performance in a graphical environment will be covered later, but first how the COMBINED organization is achieved must be covered.

For a display memory which is 1024 by 1024, with 16-bit HORIZONTAL organization, it is obvious what one method of organization could be. If 16 memory chips, which are organized internally as 65,536 by 1-bit, are used for a single plane, the bit at column H, row V is the bit in chip H MOD 16, at internal address V DIV 64. For a multiple bit access at an arbitrary location, it is relatively simple to compute the bit address.
in each selected chip, and which chips are selected. This **HORIZONTAL** organization is the basis for the **COMBINED** organization.

The **HORIZONTAL** organization has the same chip addressed from the top to the bottom in any one column. A vertical set of bits cannot be changed in one operation because multiple bits within the same chip would have to be accessed. Take this **HORIZONTAL** organization and leave the top row alone. Rotate the second row by one bit to the left, the third by two, and so on. **Done** for a 16 by 16 memory with four chips the result would be that seen in figure 5.58. (This small size was chosen for simplicity of the figure.)

![Figure 5.58 Display Memory Address Organization](image-url)

---

Table: Display Memory Address Organization

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.58 Display Memory Address Organization
In figure 5.58 the number in each cell above the dotted line is the chip number, while the number below the line is the address of the bit within the chip. The rotation was performed on the chip number and not the bit address. Any four adjacent pixels, either horizontally or vertically, address four distinct chips. With the full size memory any 16 adjacent pixels address 16 distinct chips. All discussion here will take place with relation to the small organization in figure 5.58 for convenience. Only scaling is necessary to go to the full size. If the top row is looked at, the memory appears to be organized for HORIZONTAL operations. The left column looks like it was organized for VERTICAL operations. Many of the other rows and columns look like exactly what they are, organized for X operations, but not quite lined up the way they should be.

The first thing to address is how the appropriate chips are selected. If all four chips are to be modified there is no problem. If there are less than four, which are selected depends on the address of the first bit. Consider a word which contains one bit per chip. If there are \( N \) pixels to change, the \( N \) least significant bits are set to one and the rest to zero. Looking at figure 5.58 it does not matter whether a row or column is to be modified. The leftmost and topmost pixel has one chip number. The "next" is one more, modulo the chip count. If this chip mask is rotated left by the sum of the row and column addresses of this first pixel, the result has the correct bits set to select the correct chips. For example, if three bits are to be modified starting at row 1, column 1, the original chip select bit mask is 0111 and rotating it twice gives 1101 implying that chips 0, 2, and 3 are to be selected. Looking at figure 5.58 this is indeed the case. Because the pattern of chip placement repeats every four cells the sum of the row and column addresses need be computed modulo the number of chips. Which bit within each chip to address is a much more difficult task.

The inputs available are the row and column address of the first bit (\( R \) and \( C \)), the number of chips (\( N \)), the amount that the chip mask was rotated by (\( ROT \)), the number of bits per chip per row (\( BPR \)), and the number of the chip itself (\( I \)). Single pixel modifications address the bit \( R \times BPR + C/N \). This is easy to compute, as it consists of routing wires. This expression is fine for multiple horizontal pixel modifications as well, provided the set of pixels does not cross the dark lines shown in figure 5.58.

If the multiple pixel modification crosses a dark line, there is a small problem. Investigation shows that for multiple horizontal modifications, the exact formula should be:

\[
R \times BPR + (C + (I - ROT) \mod N)/N
\]
The new factor introduced handles the step in addresses when the set of pixels cross one of the discontinuities. If a chip is the \( k \)'th bit in the set, its address is as if the column was \( k \) larger. The argument behind the derivation of the above formula goes something like this:

"If there was no rotation, I could increase the value of \( C \), for me, by my number. This would get my ‘real’ column number and everything would be fine.

Unfortunately there is a rotation, so if I am in a row which was rotated \( \text{ROT} \) left, my ‘real’ column number is more complex. I have rotated within my neighbourhood of \( N \) pixels so if I ‘fall’ off the left, I sort of ‘fell’ in the right. I can subtract \( \text{ROT} \) from my chip number to get my ‘real’ column, provided I do not fall off the left. If I do, I have to add \( N \) back to get my ‘real’ column. Rather than testing and making a conditional addition the modulo operator will take care of this for me."

The above formula works well for multiple horizontal modifications. The multiplication, division, and modulo operations are all with respect to powers of two so shifting and masking, implemented in hardware by selective wire routing, is very inexpensive. One addition is a bitwise or operation which can be done by selective routing. An expensive operation is the subtraction. For the small example it involves two, two-bit numbers. A combinational circuit to generate the correct result is very small. For the full size board the circuit has to generate a four-bit result from two four-bit inputs. For each chip one of these inputs is fixed because it is the number of the chip. If a small section of memory is initialized at manufacture time to contain, in location \( j \), the value of the chip number minus \( j \) modulo \( N \), this reduces down to one two-bit number changing into another two-bit number for the small example, and one four into one four for the full size display. The other addition is a full size addition. For the small example a four-bit adder is needed. For the real display this is a ten-bit adder.

As “luck” would have it, for multiple vertical modifications the formula is:

\[
(R + (I - \text{ROT}) \mod N) \times \text{BPR} + C/N
\]

which looks remarkably similar to the multiple horizontal formula. The arguments for its derivation are remarkably similar to those for the horizontal case.

Built as a support chip, the \textbf{BIT ADDRESS GENERATOR (BAG)} would contain one ten-bit adder. The choice of horizontal or vertical modification would determine
whether the row or column input was modified by the adder. Selected bits of the row and column values would then be routed to the outputs to provide the bit address within the chip. One BAG would generate the address for one memory chip for all planes. Given 16 chips is the full display per plane, and 16 planes, there would be one BAG for each 16 memory chips.

To move a number of bits from one location to another requires a read operation followed by a write operation. When the bits are read from the memory chips they will usually not be in a convenient position in the 16-bit word, due to the rotation imposed. The word read can be rotated left by ROT to make the bits reside in a convenient position. When placing the bits back into the memory they have to be rotated to the right by a different ROT, where this ROT is determined by the row and column addresses of the destination.

Moving bits from one location to another is trivial. To perform logical operations on these bits, while being moved, is an advantage. Figure 5.59 shows the other support chip needed for the display memory.

![Diagram of Plane Modification Processor](image)

**Figure 5.59 Plane Modification Processor**

The PLANE MODIFICATION PROCESSOR (PMP) is responsible for providing all sixteen logical operations possible between two bits, for all sixteen bits at a time, taken from the display memory. When a data word is read in from the display memory, it passes through the BARREL SHIFTER into the DATA REGISTER. The old contents of the DATA REGISTER are forced into the SECOND DATA REGISTER. The required logical operation is performed on the contents of the two registers and placed back in the DATA REGISTER. When written to the display
memory, the word passes from the DATA REGISTER, through the BARREL SHIFTER, and back to the memory chips.

Figure 5.60 shows how the memory chips, PMP, and BAG fit into the area reserved for the display memory. Depending on the final form factor of the boards, the memory and PMP for the planes can be stacked if necessary. Apart from the data flowing from the memory to the PMP and back, and the plane select signal, all pins on these chips are given the same signals on all planes at any one point in time.

![Figure 5.60 Display Memory Layout](image)

Earlier figures gave some results for this organization when used as a text display. The extra complication of having to deal with the rotations gave some benefits but may still appear to be excessive. Graphical uses are worth looking at. The first graphical use is shown in figure 5.61. Conveniently, it also shows the results of comparing various organizations. The horizontal columns represent being able to modify 1, 2, 4, 8, 16, or 32 bits in one operation. The back row represents results from a HORIZONTAL organization. The centre row shows the results from a VERTICAL organization. The front row shows the results from the proposed COMBINED organization. The left row represents the PIXEL organization where only one bit per plane can be accessed at a time. The vertical scale represents 50,000 accesses for each mark.
The program which generated the graph draws the tops of the boxes using horizontal lines, the right sides with vertical lines, and uses a block operation for the fronts of the boxes. This program was not created for this set of tests but is, rather, a small part of some other work which was being carried out at the time. The data displayed was obtained by initially drawing an arbitrary graph, and recording the number of accesses each of the organizations would use to display the graph. This new set of numbers formed the input for the second graph. This process was repeated until the numbers stabilized. The VERTICAL organization is slightly better than the HORIZONTAL because of the exact shape and size of the boxes. The proposed organization required 21,985 accesses to draw the graph. Assuming a memory cycle time of 250 nano-seconds, which is very conservative, this would take less than 5.5 milli-seconds. The best that either of the other two organizations could achieve with a length of 16 bits was 84,143 (21 milli-seconds) making the proposed organization appear to be nearly a factor of four better. Figure 5.61 may be considered a special case because it so heavily makes use of horizontal and vertical lines. Figure 5.62 shows the results for an example for which this is obviously not the case.

![Figure 5.61 Self Referential Data](image)

Returning to consider the square organization for a moment, it will be noted that both figure 5.61 and 5.62 show no mention of it. For the data in figure 5.61, to generate the picture in the chosen manner would have been grossly unfair to the square organization.
because it would perform poorly with horizontal and vertical lines. An attempt was made to contort the graphing program to favour the square organization. Sad to say “contort “is too mild a term. Dividing the parallelograms into three sections, two triangles and a rectangle is possible, decomposing the triangles into four unit wide strips, each a rectangle and a triangle does work. The square organization is better than either of the HORIZONTAL or VERTICAL organizations, barely. Producing the program to favour the square organization was interesting, but could by no means be interpreted as an intuitively simple solution to the problem of drawing a graph.

![Figure 5.62 Globe Display Data](image)

For the program generating the data for figure 5.62 implementing the distortions necessary to fully utilize the square organization was not even attempted. What would have to be done to support the square organization would be to mimic a seven by seven region of pixels centred about the first pixel not in the old square, increasing the bounds of the affected area from a one by one rectangle as each pixel was identified, until a new pixel would force one of the dimensions above a length of four. Then and only then could “one” access be used to update the display. Contrasted with what was needed to support line grouping, the overhead can easily be seen. For line grouping, all that was required was not updating the display until the horizontal or vertical run was exhausted. When that happened groups of sixteen pixels were set at a time until there were less than sixteen, then the
remainder were done with one more access. Only minor modifications to a line drawing algorithm were necessary. The resulting algorithm appears more efficient than the original. The complexity of attempting to support a square organization voted so strongly against it, that it was dropped from further consideration.

The data used to generate the results in figure 5.62 come from a program which displays the globe as viewed from any height, over any point on the earth. The land masses are shown in outline form rather than as solid colours. The original data came from four tapes found in the machine room, containing 40 megabytes of data describing the world (the fifth tape with North America was missing). This was compressed down to a little under 3 megabytes for convenience then used by the program as input. Pictures taken of the display show that there is very little that can be considered to be geared toward horizontal and vertical lines. This is evidenced in figure 5.62 by the fact that increasing the maximum modification size beyond eight pixels brought almost no improvement. Even with this pixel oriented data, the proposed scheme made 94,883 accesses (23.7 milli-seconds) against its nearest competitor with 166,755 (41.7 milli-seconds) for a difference of almost a factor of two.

There is one other complication which must be mentioned. When data is extracted from the memory for display updating, it has to go through a shifter to rotate the bits back to their “proper” place, before being used as an index into the colour map for final conversion. This rotation will take a small amount of time, but will be absorbed by the time spent in the previous word being passed through the colour map. This rotation value is based on the row number alone, and for the full size display needs to involve only the lower four bits of the row number.

Given the major improvement in accessing frequency, and the corresponding improvement in performance, it is worthwhile noting exactly what has to be done to provide a COMBINED organization after a HORIZONTAL organization exists. This is a valid comparison because there are commercially available HORIZONTAL organizations. Both organizations need to rotate the values read from the memory, and written to the memory, if access to arbitrary locations is to be supported. They both require one functional processor per plane to generate the logical combinations needed. The HORIZONTAL organization need not rotate the values used to index the colour map, nor does it need to generate the partial result 1-ROT when working out the bit address within the memory chips. These two
expenses of the COMBINED organization seem a small price to pay for the improved performance.

The means provided for access to the display memory from the PE needs to be looked at. It is all well and good to discuss the display memory as a divorced entity, but at some point it must become part of a whole. Figure 5.63 shows the twenty memory locations which the display memory exposes to the PE. Four of these twenty are used constantly for all accesses. The other sixteen are only of use in certain situations.

The use of the LENGTH MASK and PLANE SELECT locations is obvious. If six bits are to be accessed, the mask is set to have the lower six bits set to ones and the upper ten set to zeros. This is the mask that gets rotated within the display memory to select the required chips. The value placed in the PLANE SELECT location has the bits set to ones for those planes which should be affected by any modifications, and zeros for those which are to remain constant.

While these two locations are used for every access to the display memory, they tend to remain static over extended periods of time. The general pattern is that they get set, are used for a sequence of accesses, then set to another pair of values for the next sequence. Of the two, the LENGTH MASK is by far the most active in terms of change because it reflects the number of pixels to modify at each access.

The other two locations which see constant use are the last two, containing, among other things, the address of the location of the first bit in question. The use of these two locations for the row and column number is obvious. The other two uses of these locations need a little more discussion.

One basic fact which can be stated is that when the column number is written into the location assigned, this is taken as an indication that the display memory is to be accessed. Writing this location triggers the operation. The upper four bits of the location controlling the row number is used to convey the logical operation to be performed on the contents of the two data registers in the PMP. The lower twelve bits are free for the row number. While more than needed, it provides room for future extensions to a higher resolution display.

The upper four bits of the location containing the column number are the bits which control what happens within the memory. The most significant bit, when a one, indicates
that the memory is being read, and that the PMP is to load the data from its input lines. The
next bit, when a one, indicates that the memory is being written, and that the PMP should
place the value in its data register on its output lines. These two bits specify completely what
is to happen within the display memory. The other two bits appear at first to be
unnecessary. A convenient means of placing known values at known locations is needed.
There is a need to be able to determine what values are stored where.

<table>
<thead>
<tr>
<th>$70000000</th>
<th>LENGTH MASK</th>
<th>PLANE SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$70000002</td>
<td>PLANE 0</td>
<td>PLANE 1</td>
</tr>
<tr>
<td>$70000004</td>
<td>PLANE 2</td>
<td>PLANE 3</td>
</tr>
<tr>
<td>$70000006</td>
<td>PLANE 4</td>
<td>PLANE 5</td>
</tr>
<tr>
<td>$70000008</td>
<td>PLANE 6</td>
<td>PLANE 7</td>
</tr>
<tr>
<td>$7000000A</td>
<td>PLANE 8</td>
<td>PLANE 9</td>
</tr>
<tr>
<td>$7000000C</td>
<td>PLANE 10</td>
<td>PLANE 11</td>
</tr>
<tr>
<td>$7000000E</td>
<td>PLANE 12</td>
<td>PLANE 13</td>
</tr>
<tr>
<td>$70000010</td>
<td>PLANE 14</td>
<td>PLANE 15</td>
</tr>
<tr>
<td>$70000012</td>
<td>FUNC</td>
<td>ROW NUMBER</td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td>COLUMN NUMBER</td>
</tr>
</tbody>
</table>

Figure 5.63 Display Memory Access Locations

It is possible, by the use of the logical functions, to generate a zero or a one value in
any pixel location on any plane. There is no “need” to be able to provide a bit pattern from
an external source, but painting characters would consume a large number of accesses. If
the third of these four bits is a one, the contents of the externally visible PLANE X
locations are used to replace the values normally read from the display memory, and the
PMP again loads the data from its input lines. This is the first use seen of the other sixteen
locations. They provide a means of setting arbitrary bit patterns into arbitrary locations.

The last bit, when a one, implies that the PMP should place the value in its data
register on its output lines, but rather than having this information go to the memory chips,
it gets recorded in the sixteen locations which are accessible to the PE. There are two main
reasons for providing such functionality. The most obvious is that it provides a means of
obtaining the contents of the display memory so that a hardcopy can be produced, or stored
in a file. The second reason is that it provides a means of inter-plane operations. For
example, using this feature all bits on planes 3 and 4 can be inverted, provided the
corresponding bits on planes 5 and 6 differ from each other. This could be done with the
five steps:
Read planes 5 and 6 into the accessible locations
XOR the two locations and store into the locations for planes 3 and 4
Write the data in the locations into the PMPs
Read planes 3 and 4 into the PMPs using the INVERT IF SET function
Write planes 3 and 4 back into the memory
This gets repeated for each block of bits until the complete region has been processed.

Such inter-plane operations sound interesting but seem to lack many useful applications. A normal aspect display is not square. Of the 1024 by 1024 pixels, about 1024 by 800 are visible. This leaves 200 by 1024 with little use, because they are not visible. This area is typically used to store some number of pre-loaded fonts so that painting characters can be done by block moves rather than individual setting and clearing of locations. Consider that all characters are painted only on plane 0. Only one sixteenth of this extra memory need contain the font being currently accessed. If the other fifteen planes of this extra memory are used for font storage, with plane 0 of this extra used for the active font or fonts, up to sixteen times the number of fonts can be loaded into the display memory. This means that changing fonts, amongst the more popular fonts, can be done mostly within the display device, with a corresponding decrease in bus usage and hence bus contention. For typical fonts a rectangle 16 by 10 should be sufficient for the average character. Approximately six fonts can be stored on each plane. The display memory can hold up to 80 different fonts without making permanent use of plane 0. This would tend to cover most fonts in common use, for a reasonably long period of time.

Now that the display memory has been covered the OUTPUT INTERFACE MEMORY section can be looked at again. It was left earlier at the point where the contents of the buffer provided needed to be discussed. Enough background is now available to make that a viable exercise.

5.3.4 Output Interface Memory Contents

The buffer provided to the interface device contains a description of how the display memory is to be used. The first location has already been defined as the length of message indicator. The second location indicates what this message is all about.

There are four basic things that can be done with the display. It can be dumped for storage or printing, loaded with provided values, used as a character display, or used as a graphical display. The last two can be synonymous, provided the appropriate font is
available for use as the source of a block movement operation. The fourth use can be re-defined as loading an appropriate font from one of the other planes to plane 0.

If the second location indicates a loading of provided values, the next location indicates the plane which is to be affected, and the following four locations define the top, left, width, and height of the rectangle to be loaded. It is assumed that a vertical strip of memory is to be affected, no more than 16 pixels wide. A sufficient number of locations follow containing the required data. This provides a basic means of loading fonts, as well as performing such tasks as displaying stored pictures. If the plane specification location specifies no plane, it is assumed that the colour map is being addressed, and the next location provides the index of the first location affected in the colour map, then pairs of locations following provide the value to set the colour map entries to.

If the second location indicates a dumping of the display, the next five locations have the same meanings as with loading provided values. The specified plane is read rather than written. One difference is that the PE will write the requested data rather than reading it. The OUTPUT INTERFACE MEMORY is accessible for reading or writing from both sides. No reading of the colour map is envisioned. While providing this is possible, there seems little need, and supporting such an operation would have serious implications.

If the second location indicates a font change, the next five locations have the same structure as with the previous two requests. One difference is that the plane specified indicates which to read rather than which to modify which is implicitly 0. The font will be moved from the source plane to the destination plane, maintaining the same coordinates.

The previous three uses fall more in the "housekeeping" area. The last use is what the whole device is about. When the second location indicates graphical operations, the rest of the buffer contains values which can be treated as "instructions" to a graphical processor. These instructions have a four-bit "opcode" and a twelve-bit operand field. Figure 5.64 shows the four instructions currently defined.
Internally there are three sets of X and Y registers. Supplying a new value for the X register causes the contents of \( X_2 \) to move to \( X_1 \), \( X_3 \) to move to \( X_2 \), and the new value to be placed in \( X_3 \). The same applies to the Y registers. These two, with the null operation are trivial. The last instruction is by far the most complicated.

This instruction causes modification to the display memory. The second four-bit field, the Function, is the logical function to be applied in the operation. It is passed to the PMP. The last four-bit field, the Pel Set, indicates both whether or not there is an immediate 16-bit value following, and if so, what do do with it. If the value is present, it is either stored into each of the sixteen plane data locations shown in figure 5.63, or it is considered to be a pixel value. If it is a pixel value, each bit is used to determine whether the corresponding plane location is to be set to all zeros, or all ones. Used in this second manner it is possible to supply the exact value for a single point in the display. Used in the first manner it is possible to, combined with appropriate Function settings, affect a subset of the planes selected.

The third four-bit field, the Mode, is used to indicate the operation requested. The majority of operations use the contents of the sixteen plane locations of figure 5.63. The one which does not is a BLOCK MOVE, which moves the contents of the block with the upper left corner defined by \( (X_1,Y_1) \) to the block with the upper left corner defined by \( (X_2,Y_2) \) and of width and height \( (X_3,Y_3) \). Another seven Mode operations are shown in figure 5.65.
Four of these seven are area fill operations. These four are actually eight operations because they are also available as outline operations. Two others are line drawing operations, and the final operation modifies only one pixel. The area fill operations are provided for two reasons. One is that it simplifies the task of processes which are generating the graphical commands. The second, and most important reason, is that it reduces the number of instructions which need be placed in the buffer, lessening bus bandwidth requirements from initial process all the way to the display device. This is a sufficient reason to make area filling within the display device a requirement.

The three operations which work with sections of an ellipse are the most complicated to implement. They require the solution of a quadratic equation in two variables. Fortunately four points on the ellipse and the centre are known, so this is not in general a difficult problem considering that exact solutions are not required, because the display is discrete and not continuous.

One area which must be addressed by the person implementing the code which supports these operations is that cunning methods used to compute the pixels which need to be modified, must be generally known to all processes. It matters little to end users what happens provided it is fast and accurate. Problems can arise when the fast algorithms used within the display device are not known to the programmers of the applications. This can affect the accuracy that end users see. Consider the situation where a circle is to be drawn, and a line then drawn from the circle to some external point. It the programmers of the
application cannot “know” which pixels will be used to display the circle, one of the end points of the line will not be defined. If the programmers have to guess where the circle’s edge is, the line may not exactly touch the circle. Any algorithm used to interpolate these pixels must be provided to application programmers in some form. That form need not be the most efficient, provided it generates exactly the same points.

Of the sixteen possible modes of operation, twelve are defined. The other four are left for future consideration. The set chosen was picked with two considerations in mind. One, the bus bandwidth problem, has already been discussed. The other consideration was that there is only one display device processor, but multiple general purpose processors. It was conceivable that the display processor could be given the ability to perform hidden surface analysis with shadows and shading from both point and diffuse light sources, but that would have gone a great way toward constricting the bottle-neck for all. As it stands, drawing something like the wire cube in figure 5.66 takes 26 “instructions” while the surface version takes 33.

![Figure 5.66 Sample Perspective Cubes](image)

Other instructions beside those four listed in figure 5.64 are possible. For example, one can be used to allow various cursors to be loaded, and another can be used to specify which cursor is to be displayed. Such things are minor and can be ignored at this point. It is sufficient to note that there is sufficient room in the definition to allow for these and other tasks.

5.3.5 Other Parts

The PE is conceived to be another basic processor like all the rest. There is no need for an MMU or a CACHE with this processor. The program it uses, and the data locations
it accesses all reside on the display board. The only complex task it has to perform is the solution of a quadratic equation when elliptical operations are requested. Future enhancements may change this.

For example, it is an easy task to create a display board which uses a Z-buffer algorithm to support hidden surfaces if desired. Since the processor is a basic processor all of the code for the new display can be tested in a general processor until satisfied then moved to a display board with the bodies of a few subroutines being replaced with ones which do the work rather than communicating with the old display processor. Should it become necessary, even new co-processors can be designed and integrated to increase the performance of the display processor for intensive work.

![Diagram of Colour Map and Cursor Control](image)

**Figure 5.67 Color Map and Cursor Control**

The Colour Map changes 16-bit input into 24-bit output with eight bits of each primary colour. Data from the display memory, on the way to the colour map, is modified by an “and” mask and an “or” mask, as shown in figure 5.67. These two masks provide an efficient way of supporting certain operations. For example, a common instrument in astronomy is the blink microscope, used for detecting motion of planets and stars. Two pictures are taken separated by some period of time, aligned so that very distant and hence "stationary" features match, then the two pictures are displayed one after the other in rapid succession. If the display memory holds one picture in eight planes, and the second in the other eight, rapidly changing the two mask registers will cause this apparent blinking. After the application of the masks the data goes into the small section which determines whether to display the cursor, or the data. The final value is used to address the colour map.
The Display Device provides the means for a high bandwidth interaction with the human user. The use of a general purpose processor means that all code for the display processor can be checked completely by being executed as another process before committing it to the display processor. The unique organization of the display memory allows efficient horizontal, vertical and rectangular operations by supporting the most appropriate apparent organization on an access by access basis.

5.3.6 Process Support

The user interface device is supported by a small set of processes. Figure 5.68 shows these three processes. The areas named RWS are the Read/Write shared segments. The arrows represent messages sent. Each is labeled with a unique letter which will be used when the contents of these messages are described.

The IDH is created by the kernel as the process to manage the input device of the user interface device. Initially it goes through a delaying tactic until the process which will register as the UIM has done so. It then sends a type A message to that process stating that zero units of input have arrived. This is done to give the UIM a chance to assure that the user interface device is fully initialized before normal use is attempted.
After initialization the IDH goes into a typical worker loop, touching the hardware, waiting for the hardware event, then sending a type A message to the UIM to inform it that input has arrived. The message exposes the IDH's RWS for reading, giving the UIM access to the new units of input from the device. These messages are always sent to the process which is currently registered as the UIM process. Type A messages consist of only MORE_INPUT_ARRIVED requests.

The ODH is created by the kernel as the process to manage the output device of the user interface device. Its task is slightly more complex than that of the IDH. It first creates the process which will act as the UIM, then sends the newly created process a null message, to allow the UIM a chance to initialize.

When the initial message is replied to, it goes into its typical worker loop. It sends a type B message to the process which is registered as the UIM. A type B message consists of MORE_OUTPUT_PLEASE requests. When the reply to the message is provided it turns and waits for the output done event, then loops back to send the type B message to the UIM. Because the UIM is assumed to have dealt with the hardware itself the ODH need only wait for the done event from the hardware.

The UIM/WC process registers both as the UIM and as the WC. The UIM is the process that the IDH and ODH send to. The WC is the process that all other processes send to. Such a split provides a means of temporarily replacing the UIM. The UIM/WC presents a controlled windowing display. There are certain applications which do not match this model. The UIM/WC can be sent a request which states that it should "step aside". The process sending this message can register as the UIM and assume the appropriate responsibilities. The UIM/WC will wait until another message is sent by the process relinquishing this responsibility, or the other process terminates. At this point the UIM/WC can again resume the UIM tasks.

Type C messages consist of INPUT_PLEASE requests. These requests indicate a specific window. The UIM/WC assumes that no more than one process will request input from any specific window at any given time. It records the fact that the requesting process wants input from the specific window, and when there is input for that window, and a requesting process, it passes the input on.
Type E messages consist of `OUTPUTAVAILABLE` requests. These requests indicate a specific window. It records the fact that the requesting process has output for the specific window. When the ODH is waiting, and there is some process with output to be sent, it translates the output from the requesting process to output to the device. Each window has a coordinate system and this translation is essentially changing window coordinates to screen coordinates.

Type D messages are messages which are not any of the other types. The request of the message determines what the type is, rather than using the identifier of the process to determine the type. This is useful for many reasons. One example can be seen in areas such as user training. A "script" of input units can be stored in a file. A process can read this script and "pretend" to be the IDH, and feed the input to the UIM/WC at the appropriate times. The user can sit and watch (and possibly listen) as input from the pointer and keyboard do things to the window.

Type D messages are sent to the WC registered process, and include requests such as `CREATE_WINDOW`, `SHOW_WINDOW`, `HIDE_WINDOW`, `STEP_ASIDE`, `DESTROY_WINDOW`, etc. The details of how to handle each of these requests is beyond the scope of this thesis. Not only that, but the details are open to modification. Replacing the WC section of the UIM/WC can result in a different interface appearance. One version can support overlapping rectangular windows. Another version can restrict windows so that overlapping is not allowed.

Section 5.4 The Disk Device

The Disk Device appears rather conventional. Figure 5.69 shows a rough version of what the board would look like. Its appearance on the bus is much like that of the User Interface Device. The BUS MASTER section is used to inform the kernel processor of the completion of an operation, and the BUFFER DEVICE section is used for communication with the interface process. The DISK BUFFER MEMORY is used as a convenient cache of recently accessed disk blocks to increase the apparent speed of disk operations.

The detection of the interface process having requested an operation is handled in exactly the same manner as with the User Interface device. When the first location of the buffer is written from the bus, the Processing Element is able to determine that it is to now
start its processing. The contents of the buffer consist of two pieces, the first is a message indicating what has been requested, and the second is a data section.

The contents of the message will be familiar from the discussion of the operating system. It is almost the same message as was given to the numbered file system process. The numbered file system process is a front end process to this device. The Disk Device is a file system device. The range of requests in this message is greater than that to the file system process, in order to support the other operations required to manipulate the storage media.

![Diagram of the Disk Device Board]

Figure 5.69 The Disk Device Board

All permanent disks physically connected to the machine are considered to be parts of one large file system. In general, the use of disks to store the file system is transparent. All accesses are done with reference to a file number. Where and how that file is stored is not important.

There are sixteen special "files". None of these files are within the normal file system. They provide a "lateral" access to the file system storage, and other devices. The file with number $FFFFFFFC$ is the "file" which contains the configuration information. It indicates what disks exist, their sizes and positions within the full storage space, and the physical sector length in bytes. The sizes and positions are based on the assumption that
blocks on the disks are 4096 bytes in length. Up to ten disks can be attached, and have "file" numbers from $FFFFFFF0 to $FFFFFFF9. There is always a need to descend to disk operations at some point, even if to perform full back-ups and restores in an efficient manner. These "files" provide the ability to do so as needed.

File $FFFFFFFB is considered the back-up device associated with the file system, if such a physical device exists. Not all machines will be configured with one of these devices. For a network of machines, for example, possibly only one will have a large scale back-up device. All others will make use of it. For isolated machines the cost of such a device will probably imply that this "file" does not exist.

File $FFFFFFFF is considered a removable disk of some kind. It will probably be a floppy disk. All machines need not have one of these "files". Isolated machines may have this "file" for software distribution and local back-ups. Networked machines will probably not, in general, have this "file". Since this "file" is considered a temporary one, it does not form an integral part of the file system.

File $FFFFFFFA is considered an archiving device, such as a WORM disk. It is meant for very permanent storage. The inherent features of such devices make them unsuitable for integration into normal file systems. Not all machines will have such a "file".

The other two "files", $FFFFFFFD and $FFFFFFFE can be used as desired. They can, for example, be configured to support another device of a back-up, floppy or archiving nature.

Initially only these special "files" will be referenced. Using these, the storage of a sufficient number of file descriptors can be done so that, from that point on, the other files are available.

The request written to the device indicates which "file" is being accessed. There will be an event generated in response to every request. Requests which require storing information will be accompanied by data in the buffer area. Requests which require loading information will be responded to with data in the buffer area. To allow overlapping operations, because accessing physical devices take time, read operations are composed of two operations. To read X units from file Y at location Z, the first request specifies that the device is to go and get the information. This will eventually generate an "I found it" type of event. The second request specifies that the device is to provide the information.
The buffer area consists of three parts. There is the area into which the interface process places a request, an area from which the interface process can read a response, and a large area through which data passes. The response returned by the device is the request it was given, with a status indicator included. As with the user interface device, the processing element can detect when the request and response areas are available.

A storing request will result in a response as soon as the data area of the buffer is emptied. A request that asks for data to be loaded will be responded to when the information is loaded from the device and is waiting to be picked up. A request that asks for the loaded data to be provided will be responded to as soon as the loaded data is placed in the buffer area. There can be many storing requests which are not yet complete at any point in time, along with many loading requests.

The processing element on the board, another of the basic processors, is in control of its local buffering memory, and the ordering of requests to the physical devices. If the physical disk controllers support overlapped seeks, this feature can be used. Various request queue ordering algorithms can be implemented. By having a processing element which runs a program that is tuned to the exact hardware available, the best use of that hardware can be made.

![Diagram of Disk Device Processes](image)

**Figure 5.70 The Disk Device Processes**

The kernel will create the **Disk Interface Controller** process and give it the ability to address the device address space. This process creates two more. One is the **Event Waiter**. The other is the **Access Controller**.
The **Event Waiter** is a very trivial process. It repeatedly waits for an event, the number of which it was given by its creator, then sends an `EVENT_HAPPENED` request to its creator.

The **Access Controller** is the top level process in the file system. It "knows" that its creator is the disk device process, and can tell all other processes, which it creates to manage the file system, that this is the process to talk to.

The **Disk Interface Controller** sits waiting until a message is received. If the message comes from the **Event Waiter**, it can read the response area of the device interface to determine what the response is referring to. This will allow the **Disk Interface Controller** to complete the task that was requested. Should the message come from some other process, actions are somewhat more complex.

Only one request to write to the device can be outstanding at any one time, assuring that the device will never run out of buffer space. If there is space, the device processor will buffer the data, and respond immediately. If space is not available, it will hold off responding until there is space. While multiple writes to the storage medium can be handled at the same time within the device, this rule assures that there will never be a point where the device's buffer space is exhausted.

The **Disk Interface Controller** is capable of holding a limited number of operations in progress. When all of its entries for holding requests are filled, it switches to receiving from only the **Event Waiter**. This assures that no future entries are needed until at least one is free.

**Section 5.5 The Memory Device**

Memory presents a difficulty in a multiple processor machine. The basic problem with such a machine is that there will inevitably be contention for the bus. If reads are done with a very wide word, accesses for instructions need only be made every few instructions. If enough data is read for 16 instructions with each access, 16 processors would place the same load on the bus as a single processor, getting one instruction per access. The other side of the coin recognizes that a complete memory cycle extends past the end of the effective access time. For example, when writing a word, the memory can acknowledge the write before the bits have settled, and long before the accessed chips are able to respond to another request. This leads to a perceived need to interleave the memory, so that while one
location is “recovering” from an access, another can be responding. Merging these two concepts results in memory which appears as an extremely large number of sections of extremely long words.

Such a concatenation of two extremes is viable, if a machine with a 524,288 addressable units of memory has to be constructed out of memory chips which each contain only 1024 bits of information. It would take 8192 chips in any event, so they might as well sit side by side rather than end to end. There could be 64 parts, each providing 128 bits of information. Doing exactly the same thing with common chips now available, with 262,144 bits each would result, with the same interleaving and access size, in a system that has 134,217,478 addressable units of memory. While adequate for most purposes, it does seem a little excessive as an entry level machine, though it would cut down on the amount of temporary file space needed.

It is interesting to note that the arguments for large access sizes and high interleaving factors are based on probabilistic arguments. The same basis exists for “proving” the usefulness of cache memory. There is no guarantee that anything will help, it is only that it probably will, because it would require a pathological program to negate the benefits.

Trying for 64-way interleaving is obviously excessive in a single processor machine, unless the memory is exceedingly slow. It is probably excessive in a multiple processor machine, even one with 64 processors. Consider the birthday “paradox”. With 365 days to choose from, gather 20 people at random and the chances that there will be a birthday clash are high enough that one should not bet against it.

The waters are muddied even more by the existence of cache memory for each processor, which is expected to absorb a large amount of the potential memory accesses.

If chips with 262,144 bits are used, and an access word size of 128 is chosen, the smallest memory possible is 2,097,152 addressable units. This defines the entry level memory size, and that the entry level memory is not interleaved. Adding another memory increment can make the memory into a two-way interleaved memory of twice the size. This interleaving can be done in a conventional manner, with the least significant bits of the address selecting the part in question, or the most significant bits of the address can be used, essentially creating two separate memory banks. The potential number of solutions is immense. Figure 5.71 shows the chosen one.
There is no cache memory on the memory device at all. The cache memories associated with the processors do their best to assure that if the memory is accessed for location X, it will never be accessed for location X again. A cache memory unit on the memory device would only hold values that would “never” be accessed again. Given that all processes have disjoint data segments, this is true. For shared code segments the argument is weaker but still a reasonably valid observation. There would be multiple accesses to a location from a single processor, if the location has “fallen out” of its cache, or if the location is being written. Since the processor’s cache holds written data as long as possible before accessing memory, some multiple accesses will be folded into single accesses. A burst of write accesses will be generated from a single processor when it is switching processes. This is the justification for the WRITTEN MEMORY (WM) unit which, despite what has just been said, acts a little like a cache.

![Figure 5.71 The Memory Device Board](image)

When a write access is made, the data is dropped into the top of the WM unit. The data is positioned appropriately within a 128-bit word. The data word falls down the WM and either is caught by some entry with the same address, or it falls out the bottom. If it is caught, that entry is adjusted, in the appropriate bits, with the new value. Should it fall out the bottom, the top entry in the WM is used to update the memory, every entry moves up one position, and the entry that fell out the bottom is pushed back into the empty location. In a situation where consecutive memory locations are being written, the memory need only be
accessed after 128 bits of data have been generated. Interleaving is normally used to support accessing consecutive locations without the need to wait for a full access cycle between accesses. Reading 128 bits at a time avoids the pause for reads, but writes do not always happen on complete 128-bit objects. Collecting of writes, when possible, goes some way toward removing the reason for interleaving.

Because the processor's cache holds writes as long as possible, there is a reasonable chance that any writes directed to the memory device will cluster, and the effective number of accesses will be reduced by this scheme.

When a read access is made, this access is passed through to the memory which provides the basic 128-bit value. Should any of the entries in the WM cover any of the read locations, this 128-bit value has to be updated to reflect the values that the memory should have contained. Because of the scheme used when values are written, there will never be more than one entry which will match the address.

The implementation of the WM unit will be familiar. The requirements of this unit lead to its implementation as a rotating ring of "registers". Each entry contains one 128-bit data section, one 29-bit address section, and one 8-bit "data written" section. The number of "registers" in the ring is an open question, the more there are, the greater the chance that two write accesses can be merged, but the longer it takes for a complete rotation. Since every entry must be examined on a read access, if the number of entries is too large, read access times will be lengthened. For convenience of discussion, let it be assumed that there are 64 entries.

When a write access is attempted the address of the accessed location is compared with the address section of each entry as the ring rotates past the write port. If a match is encountered, the new value is merged into the data section, and the correct "data written" bit (bits) is (are) set. Should the ring have shifted 64 times, the entry in the write port is swapped with the new entry, and the data swapped out is written to the address swapped out, using the "data written" bits to generate the appropriate chip select signals. The time taken for this operation is overlapped with the time used for the bus protocol, and will in all likelihood be complete before the next access can be requested.

When a read access is attempted, the address is passed to the memory, and while it is responding, the ring is rotated past the read port, looking for a matching address. If a matching address is found, the data in the entry is merged with the data obtained from the
memory, under the selective signals from the "data written" section. If the time to completely cycle the ring is greater than the read access time of the memory, the effective access time of the memory will be degraded. This can be partially overcome by using multiple read ports on the ring. If cycling 64 entries takes longer than a read access, two read ports would take half as long. With the ring guarantied to find a match, if one exists, in less than the read access time, the signals to the gates in the merging circuit will all be set by the time the data passes through from memory to the bus interface. If too many read ports are placed on the ring, the delay in passing through all the merging circuits can affect the read access noticeably so there is an upper limit on the both the number of entries in the ring, and the number of read ports.

This interaction between memory speeds and maximal ring length implies that as memory access times decrease, the ring length must decrease. At some point the maximum ring length will approach zero. That point in time is of little concern. Improvements in memory technology will have spelt the doom of cache based memory products long before then.

There is a large number of people who "believe" in virtual memory. No concession has yet been made to these beliefs, although provision has been hinted at. If the memory banks in figure 5.71 are removed and replaced with the structures shown in figure 5.72, virtual memory is supported.

![The Virtual Memory Device Board](image)

**Figure 5.72 The Virtual Memory Device Board**
When a read access is attempted, from a memory location which is not in one of the loaded pages, the bus signal discussed earlier as the **RESPONSE** signal is set to indicate a delayed response. The bus master which initiated the access would detect that it had to retry the access. This event starts a sequence of operations to have the requested page of memory loaded from backing store. Failed writes cannot be responded to as delayed, because the bus master (masters) has (have) long since been informed that the write was successful. Such faults have to be queued, and serviced, as the appropriate pages are loaded.

Supporting virtual memory is possible. The economic factors which determine what is the best choice, lots of memory, or a little memory and an expensive but large backing store, change constantly. The availability of wafer scale memory products is important because the speed differential between real and backing memory may imply that rather than indicating that the read should be retried, it should just take a bit longer. The rapid evolution of memory technology makes being more specific about how virtual memory is to be supported, worthless. Any detailed definition would be outdated long before a physical implementation of this machine would exist. The integration of virtual memory is left for future consideration An indication of how and where it would be inserted has been given.

Ignoring virtual memory, this memory board provides a wide access word to reduce the number of accesses for read operations. Using the unique features of the **WM** section this reduction in true memory accesses can even be extended to writes, despite the fact that writes are not performed on wide access words. Combined with the **pending-write** caches in the processor boards this can avoid the need for interleaving of memory.

**Section 5.6 The Communication Device**

The communications device is rather common. It bundles up to 15 serial or parallel lines into one device. It services the lines locally to reduce the number of interrupts which need disrupt the orderly processing of the rest of the machine. Figure 5.73 shows a view of this board.

The board supports communications on lines 1 through 15, reserving line 0 as a configuration control line. Output to line 0 is considered as a list of instructions to the communication device itself. Input from line 0 is generated as a list of status indicators from the communications device.
The output buffer is filled with a length, line number and sequence of values. For lines 1 through 15 these values are to be output on the specified line. For line 0 these values are used to configure the communications lines. For example, one output may specify that line 6 is to be used at 19.2 Kbaud, asynchronous communications, status reporting when input buffer has more than 37 characters or one tenth of a second has gone by, XON/XOFF protocol to be used, etc.

Figure 5.73 The Communication Board

The input buffer contains a length, line identifier and sequence of values. For lines 1 through 15 these values are input from the specified line. For line 0 these values provide status indicators of all lines.

Being so conventional there is little need to go into any great detail with this device. Its method of using the events provided by the kernel and synchronization of buffer usage should be obvious from previous descriptions with respect to the User Interface device. As far as process support is concerned, a central administrator process is supported by two worker processes which are created by the kernel when it has detected that the input and output devices exist.
Section 5.7 Other Devices

The few devices already covered should make it obvious how other devices would be incorporated into the machine. Such things as voice synthesis or video frame acquisition hardware can be designed and packaged with a section of memory containing the program for the process to support it.

The chosen means of integration of devices makes adding new ones simple. For example, if a fax unit is bought, it can be added to the machine, plugged into the telephone socket, the telephone plugged into the fax unit, and the machine turned on. The user presses the train button and dials a sample fax number. The fax unit is configured to generate pulse or tone signals as required, and follows the user's pause pattern in dialing future numbers.

Ancillary software can come on some transportable medium, and can be installed using a standard method. If the manufacturer so desires there is an alternative solution. When the machine is turned on, a process to handle the device is created. That process can check to see if the support software has been installed. If not, the support software, stored in a very cheap and very slow memory within the device, can be automatically installed. In this second method the user, after turning the machine back on, sees a fully installed fax unit complete with all software and online documentation. While this second method of software distribution is more expensive, the benefits are a marketing problem.

Section 5.8 Packaging

A basic machine is going to contain at least two processors, a power supply, a section of memory, a user interface device, and either a network device or a storage device. There will be at least five units in a machine. Most units will fit within a box which is 25 cm. by 18 cm. by 3 cm. The box that each unit is enclosed in will look remarkably like a book. When the units are placed together they will look like a set of books. The bus will pass through all the “books”. The power supply will form the last book in the set and the User Interface Device will be the first. These two units will have the connections for the bus signals which loop back to form a ring. The power supply unit contains the three stub devices needed to fully populate the address space on the bus. Figure 5.74 shows one of these “books” in both a cut-away side view and from the front so that the air flow pattern for cooling can be seen.
Figure 5.75 shows a complete machine. The power supply unit is built into the "book", and the right book-end. This provides enough volume to handle all the necessary components, and provides space for an exhaust fan used to pull air, which enters the left book-end, through the machine.

To add a new unit the user need only separate the power supply "book" from the rest, insert the new "book" in place, set its address switches to the settings of the stub devices on the power supply unit, and adjust the address switches on the stub devices as appropriate. For example, to add a voice synthesis unit the device address of the stub switches is used to set the device address of the unit, and the address of the stub unit is incremented by two. The power supply unit is then pushed back to connect it with the rest of the units.

Should a device need to be removed, the operation is slightly more complex. The address on the sub device has to be modified, and potentially other devices have to be moved in the address space as well. The simplest method is to separate the "books" at the device to be removed, after the last unit of the same type i.e. another device or another piece of memory, and finally at the power supply unit. Remove the unit in question. Move the last unit of the same type into the vacated space, resetting the address switches to those of the removed device. Decrement the stub switches as appropriate and push the units together.
Figure 5.74 shows air flow over the board in one of the units. This is the active cooling mode. If desired, the “top” and “bottom” of each unit can be replaced with a grill. If this is done, there is no need for forced air cooling, as each board is provided with passive cooling. Passive cooling is the preferred choice. It avoids the need for a noisy fan to move the air. The negative aspect of passive cooling is that the “books” may not look as nice.

A keyboard and flat display are packaged as an oversized book. When opened it may be used as both a keyboard and display. Normally it is connected by a cable carrying signals in both directions. For those who wish, the flat display may be taken off the keyboard and stored away, in which case some other type of monitor would be connected. If left together, the book may be disconnected from the machine. The keyboard and display become a “laptop” computer, which may be taken from place to place. This laptop computer is very limited in capabilities. It has one serial port, and one memory disk drive. It functions as a data entry and storage device when used as a separate unit. When reconnected to the machine the information it recorded can be integrated and more sophisticated manipulations are possible.
Chapter 6
Conclusions

The design of both an operating system and the hardware it is to run on, have been covered. This integration of design has allowed, not so much a compromise, as a symbiosis. For every problem there are a number of solutions. Taken in isolation the choice tends to be made in an arbitrary manner. By looking at the implications of each choice with respect to the other components it is often possible to make a more reasoned choice. In some cases a solution which appears less than optimal in isolation may turn out to be the best overall. This is true, for example, with memory management where the hardware solution can have the simplicity of both paging and segmentation, when it is realized that the operating system only needs some protection and mapping mechanism that requires minimal management.

The system uses a message passing scheme for communication. Isolating processes and providing a simple scheme for interaction increases the confidence level of programmers that any “features” that a process exhibits are its own responsibility. The alternate of a shared memory communication scheme, with some cunning derivative of semaphores as a control mechanism, cannot provide this. Shared memory is supported, but considered only as a means of massive data transfer in an inexpensive manner.

The send operation blocks either until reception or response depending on the “type” of the message sent. Various non-blocking styles of send operation were considered, but the supposed benefits are far outweighed by either the complexity necessary to assure correct use of such primitives or the complexity necessary to actually provide the primitives. Mailbox style primitives were also considered and rejected in favour of direct addressing to processes. The complexity to assure correctness of a mailbox scheme is excessive when the supposed benefits are actually examined. The two differing types of messages, questions and statements, lead to the decision to base the duration of blocking on the type of request sent. The kernel examines the value of the request field of the message, and uses one bit of the request to determine if the sender should remain blocked after the message is delivered. This restriction on the format of the message is not severe when messages can potentially pass between non-homogeneous machines in a network, which requires full knowledge of the message structure.
Messages can be received either from one specific process, or from any process which is attempting to send. A specific receive primitive provides the facility to simulate request code screening as well as providing an indication of the existence of the communicating process. Servers in general use the non-specific receive to accept messages from all clients.

A special means is provided for movement of massive amounts of data. Both processes involved in this movement are in control of both the area involved and direction of movement. For efficiency reasons, it is necessary to allow servers such as the file system processes to transfer data to and from client processes rather than restrict them to just data in messages. By providing the client processes with a means of rigidly defining where, how much and which way data flows, the only possible modifications to the data of a process that are not under its direct control are constrained. A process may, if desired, expose all of its data for modification, but that has to be a conscious decision.

The kernel provides support for a total of only fifteen operations. This small number are all that are needed given the hardware provided. The only complex part of the kernel is the section which determines which processor a process should be given to. Having non-homogeneous processors leads to some complication, but having various flavours of each type of processor is the major factor. An algorithm has been given which allows an efficient means of arranging that each ready process will be assigned to the processor which is the best choice overall, given the information available to the kernel.

While the use of a buddy scheme of memory allocation provides an efficient means of hardware memory mapping, it can result in inefficient algorithms for compaction when that becomes necessary. The use of a visualization scheme, to decide which sections of memory to make empty, results in a minimal amount of shuffling with only a small amount of processing needed.

Two processes are used to manage the loading of programs and creation of other processes specified by symbolic names. By reducing the kernel to handling programs by number only, and only programs which are in memory, the kernel is simplified and isolated from matters which are not directly its concern. This also simplifies the handling of the loading of programs. Because different tasks are not interwoven within the same process each can be approached cleanly and simply. As a by-product, programs not used by processes can be left in memory. They only need to be removed when the space is required.
This implies that many programs need only be loaded from the backing store a vanishingly small percentage of the times they are needed.

All file accesses go through one process which redirects them to appropriate name lookup processes. Having one process to send to for all file accesses makes programming easy. Having multiple name lookup processes allows simple integration of such things as remote file systems and device support. For example, having a file system “tree” which represents all potential printers allows a user to specify an exact printer, or a more lax definition if that is desirable. The name lookup process handling printers can use the rest of the “path name” given to determine the appropriate printer.

The use of remembered previous searches is used rather than a current directory scheme to allow rapid path name lookup. A current directory scheme works well when file accesses are restricted to one sub-tree. When accesses are scattered, only a proportion can benefit from the current directory saving. By not supporting a current directory there is no need to tightly intertwine the concept of a “position” in the file system with each process. Using a simple cache of node names provides efficient accesses to sub-trees within the file system under which repeated accesses are made. This provides, essentially, the benefits of multiple current directories. Because this cache is universal rather than restricted on a per process basis, one process can benefit by the accesses previously made by others.

This bus provides support for multiple bus masters, both occasional and constant users of the bus. Distributed arbitration based on a token passing ring concept allows the number of bus masters to vary as needed. The introduction of three “stub” devices assures that all possible addresses on the bus are “valid” and there is no need for watch-dog timers or the like. The response returned from devices can indicate a status enabling appropriate further actions by the bus master.

If a processor is idle it places absolutely no load on the BUS. The memory management method chosen removes the need for any adder circuits within the MMU. The loading of the description of a new process to execute can occur simultaneously with the continued execution of the current process. The CACHE unit can operate in a pending-write mode, and the early detection of a future process switch can be used to induce the completion of these pending writes, possibly before the new process description has been completely provided.
A communications ring inside the **PROCESSOR** allows functional units inside the **PROCESSOR** to operate in parallel. The **CLU** allows multiple non-conflicting instructions to be executed in parallel. The **IFU** provides sequential pre-fetch capabilities as well as caching of a sufficient number of instructions in an internal format that small loops can be executed without recourse to external memory or instruction decoding. As well, pairs of instructions may be folded into one to further reduce execution time. The kernel processor board is distinguished from others by one simple component. Moving this component from one processor board to another moves the kernel processor and reduces the critical component problem down to only one chip.

The integration of co-processors into the whole processor board is as “seamless” as possible. Because co-processors substitute for subroutines there need only exist one executable version of any program. The alternate “seamless” approach of having subroutines substitute for co-processors implies that the processor has to be built with the existence of co-processors in mind. The chosen method allows any co-processor of any kind to be designed and integrated at any point in the future. A faulty co-processor can simply be removed. There is no need for all processor boards to contain the same co-processors. Should the switches informing the kernel of a co-processor's presence be incorrectly set the only effect is that the kernel will make less than optimal choices of which processor to assign certain processes to.

The display memory organization for the user interface device provides the ability to address pixels in both horizontal and vertical groups. This allows a controlling program to make an intelligent choice of access arrangement which will lower the number of accesses needed to complete a given task. The organization chosen gives both efficient accesses, and simplicity. Algorithms to draw lines not parallel to either axis can still benefit, without adding complexity.

The memory units implement a “write delay” form of access. This allows the detection of multiple write requests to consecutive locations, and the folding of those multiple writes into a single write access to the true memory. This feature is obviously useful at process switching time, but also can be of great benefit when storing array entries or passing arguments to subroutines. Since multiple blocks are supported, this delayed write feature can even deal with scattered writes, holding the actual memory accesses for as long as possible.
Virtual memory can be supported, but is supported by the memory board rather than by hardware associated with processing elements and software within the operating system. This approach removes any need to wed the whole machine to some agreed upon virtual memory model. As memory products evolve and new memory boards become available they can just be placed in the machine and used. Even mixes of virtual memory boards are possible. One board might use slow bubble memory which implies that a "try later" response to the access should be given, while another may use wafer scale integration of slower memory implying that the access should just take a little longer than usual.

The ability of a software producer to distribute software in ROM has advantages. ROM software is not a new idea. Games manufacturers often produced "cartridges" with their software included, which got pushed into the machine to make it available. The best aspect of this from the manufacturers point of view was that there was no potential for piracy of software. By having a large library of routines in ROM, the user benefits by having smaller programs stored on disk and loaded into memory. Smaller programs mean more effective disk space, and more effective memory space. Implementing the library "ROM" as RAM that is loaded when the machine is started allows improvements to the library to be made which automatically appear in all programs which use the library, even third party software.

The hardware provides the ability to allow automatic creation of processes to handle devices, and removes much of the configuration problem. Adding or removing a device from a machine may require no more from the user than the addition or removal of the actual hardware unit itself.

Packaging the machine as a set of "books" makes it more esthetically pleasing, and can go some way to removing prejudices of higher executives who feel that having a "machine" on their desk lowers their status.

The keyboard and flat display in a "book" which also can function as a laptop computer provides the portability that is needed in various situations while still being able to function as an integral part of the larger machine.

All reasonably interesting unique features of the various components have been covered. Some of these features, such as the display memory organization, can stand on their own merits. Others, such as the instruction set of the processor, hang suspended in the
interrelated web of hardware and software components. The conclusions reached here have been arrived at by taking questions of the form, “How do I do ... ?” and replacing them by, “Given that I could have ... done, what is the minimum needed to have the effect, not necessarily with the cause?” as a guiding principle. For example, “How do I tell the co-processors what their arguments are?” is replaced by, “How do the co-processors find out what their arguments are?” Like a mobile, all parts must be present for the final product to be “understood”.

303