

The Implementation of Concurrent Pascal on the NCR8200

by

Donald Mounday

B. S., Fort Hays Kansas State College
Hays City, Kansas 1969

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

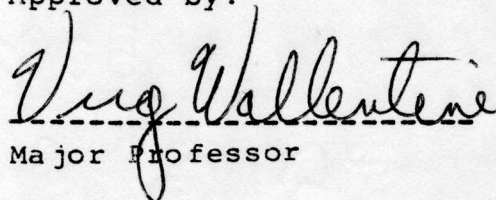
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1978

Approved by:



Major Professor

Spec. Coll.
LD
2668
.R4
1978
M68

Table of Contents

1.	Introduction.....	1
1.1	Origins of Concurrent Pascal.....	2
2.	NCR8200 Concurrent Pascal Implementation.....	4
2.1	Implementation Approach.....	4
2.2	Implementation Overview.....	5
3.	Transporting Concurrent Pascal to the NCR8200.....	10
3.1	NCR8200 Hardware Architecture.....	10
3.2	Problems of Porting.....	11
3.2.1	Byte Address Simulation and Address Segmentation...	13
3.2.2	IO Buffers.....	17
3.2.3	Floating-Point Emulation.....	17
3.3	Portability Evaluation.....	17
4.	Adapting the Concurrent Pascal Machine.....	20
4.1	Addressing.....	20
4.2	Virtual Code Modification.....	28
4.3	Adaptability Evaluation.....	34
5.	A General IO Architecture for the Pascal Kernel...	35
5.1	Driver Classification.....	35
5.2	Mapping Pascal IO to Existing Drivers.....	36
5.3	IO Request Handling.....	40
6.	Summary.....	42
	References.....	45
	Appendix.....	47

List of Figures

2.1	Concurrent Pascal Transportation.....	7
2.2	NCR8200 Concurrent Pascal Implementation.....	8
2.3	Memory Allocation.....	9
3.1	ADT IO Buffer.....	12
3.2	Byte Address Simulation and Address Segmentation.....	15
3.3	Pushing Data addresses.....	16
4.1	Push Character Array Element(byte addressable).....	23
4.2	Copy Character to Array Element(byte addressable).....	25
4.3	Virtual Code for Character Array Operations (byte addressable).....	26
4.4	Stack Pictures of Array Element Copy (byte addressable).....	27
4.5	Virtual Code for Character Array Operations (word addressable).....	30
4.6	Push Character Array Element(word addressable).....	31
4.7	Copy Character to Array Element(word addressable).....	32
4.8	Stack Pictures of Array Element Copy (word addressable).....	33
5.1	IO Levels(Pascal to Hardware).....	38
5.2	IO Device Structures.....	39
5.3	IO Request Handling.....	41

List of Tables

4.1	Data Storage Requirements.....	22
6.1	Summary of Development Effort.....	44

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

1. INTRODUCTION

Highly portable and adaptable software has long been a desired goal[PW]. Concurrent Pascal[BH1] is a programming language in which portable operating systems can be constructed. Concurrent Pascal allows the construction of operating systems with a hierarchical ordering of abstract data types. New data types have been invented by Brinch Hansen for the specification of processes and shared variables, and for the sharing of common code. This report contains descriptions of the transportation of Concurrent Pascal to the NCR8200, the adaptation of the Pascal virtual code to the addressing scheme of the 8200, and a generalized IO architecture to aid in the development of new peripheral device drivers for the system.

This report is organized in the following manner: the remainder of chapter one contains a description of the evolution of the Concurrent Pascal system that was implemented on the NCR8200. Chapter two contains a description of the implementation of Concurrent Pascal on the 8200. A description of porting Concurrent Pascal to the 8200 is presented in chapter three. The adaptation of the virtual code to a word addressable architecture is described in chapter four. Chapter five contains a description of a generalized IO driver architecture for the Concurrent Pascal system. This paper concludes with a summary in chapter six and the appendix contains the pseudo-code of the interpreter

for the adapted virtual code.

Much of this report is based heavily on work done by Neal[DN1] in transporting Concurrent Pascal to the Interdata 8/32. This report assumes that the reader is familiar with Concurrent Pascal and its runtime support requirements.

1.1 Origins of Concurrent Pascal

Concurrent Pascal was invented by Per Brinch Hansen at California Institute of Technology(cit). It was implemented on a PDP11/45 and used to develop the single user operating system, SOLO[BH2]. Two compilers, one for sequential Pascal and one for Concurrent Pascal, and several utilities run under the SOLO operating system. The compilers are written in Sequential Pascal and generate a virtual machine code that can be executed efficiently with a machine language interpreter.

The Concurrent Pascal machine is implemented with two assembly language programs, the interpreter and kernel. The interpreter executes the virtual code of concurrent and sequential Pascal programs. The kernel provides the interface to IO devices, handles device interrupts, controls access to shared variables, handles timer services, and multiplexes the processor between the concurrent processes.

Kansas State University Department of Computer Science personnel implemented Concurrent Pascal on the Interdata 8/32[DN2]. This implementation was done as a subsystem to the OS/32-MT operating system. However, the structure of

the kernel and interpreter was not changed significantly from that defined by Brinch Hansen. Some minor modifications were made to the representation of strings and sets to alleviate character string representation problems inherent with the PDP/11 addressing scheme. The I8/32 implementation required writing the kernel and interpreter in CAL assembly[DN2] language and building the SOLO virtual disk from the CIT distribution tape.

An image of the I8/32 SOLO virtual disk, containing the ASCII source files and virtual code for the SOLO system, was the starting point of the NCR8200 Concurrent Pascal implementation effort.

2. NCR8200 CONCURRENT PASCAL IMPLEMENTATION

This chapter contains a description of the implementation approach and an overview of the NCR8200 Concurrent Pascal implementation.

2.1 Implementation Approach

The implementation approach was chosen with manpower, the primary resource, in mind. Three development approaches were available:

- 1) Implement the entire system on a bare machine.
- 2) Implement the interpreter and kernel on an available general multitasking package with drivers and memory management facilities.
- 3) Implement the interpreter and kernel using EXECI[NCR1], a single user development system, for IO driver support.

The first alternative was out of the question due to the heavy manpower investment that is required to develop IO drivers for the 8200.

The second alternative seemed more attractive but a considerable effort was necessary to map the Concurrent Pascal machine task structure into the task structure of the multitasking package. In addition the device drivers were much too extensive in their capabilities for the Pascal system. They provided such features as error logging, power-fail recovery and peripheral diagnostics. The 6K words of memory required for the drivers seemed too expensive in

comparison to the features provided by the second alternative.

The third alternative was chosen because it provided debug aids, simple file management capabilities, and a virtual line printer and card reader. The major disadvantage was the IO driver architecture. Chapter five contains a discussion of the problems in this area and a general solution.

2.2 Implementation Overview

The procedures of transporting Concurrent Pascal to the NCR8200 are depicted in figure 2.1. A magnetic tape image of the I8/32 SOLO virtual disk was copied directly to an EXECI data file. The kernel and interpreter were coded in the 8200's assembly language and then linked and loaded by the EXECI operating system.

The NCR8200 Concurrent Pascal implementation structure is illustrated in figure 2.2. The kernel and interpreter were implemented on top of the EXECI operating system. As EXECI does not provide any multitasking support, processor multiplexing was implemented totally within the kernel. EXECI did provide IO support for the SOLO devices and the necessary development tools and debugging aids.

Memory allocation for the 8200 is illustrated in figure 2.3. The kernel does not contain constants for the maximum number of process records or monitor gates. This memory is allocated when requested from the bottom of the interpreter

toward the private data space of the processes. The private data space for processes is allocated from the top of the initial process and the common area in low core toward the bottom of the process records and monitor gates. In allowing memory to be allocated in this fashion the system does not have to be reassembled and configured to change the number of monitors or processes in a concurrent program.

More details regarding the implementation are presented in chapter three on portability and chapter five on the IO architecture.

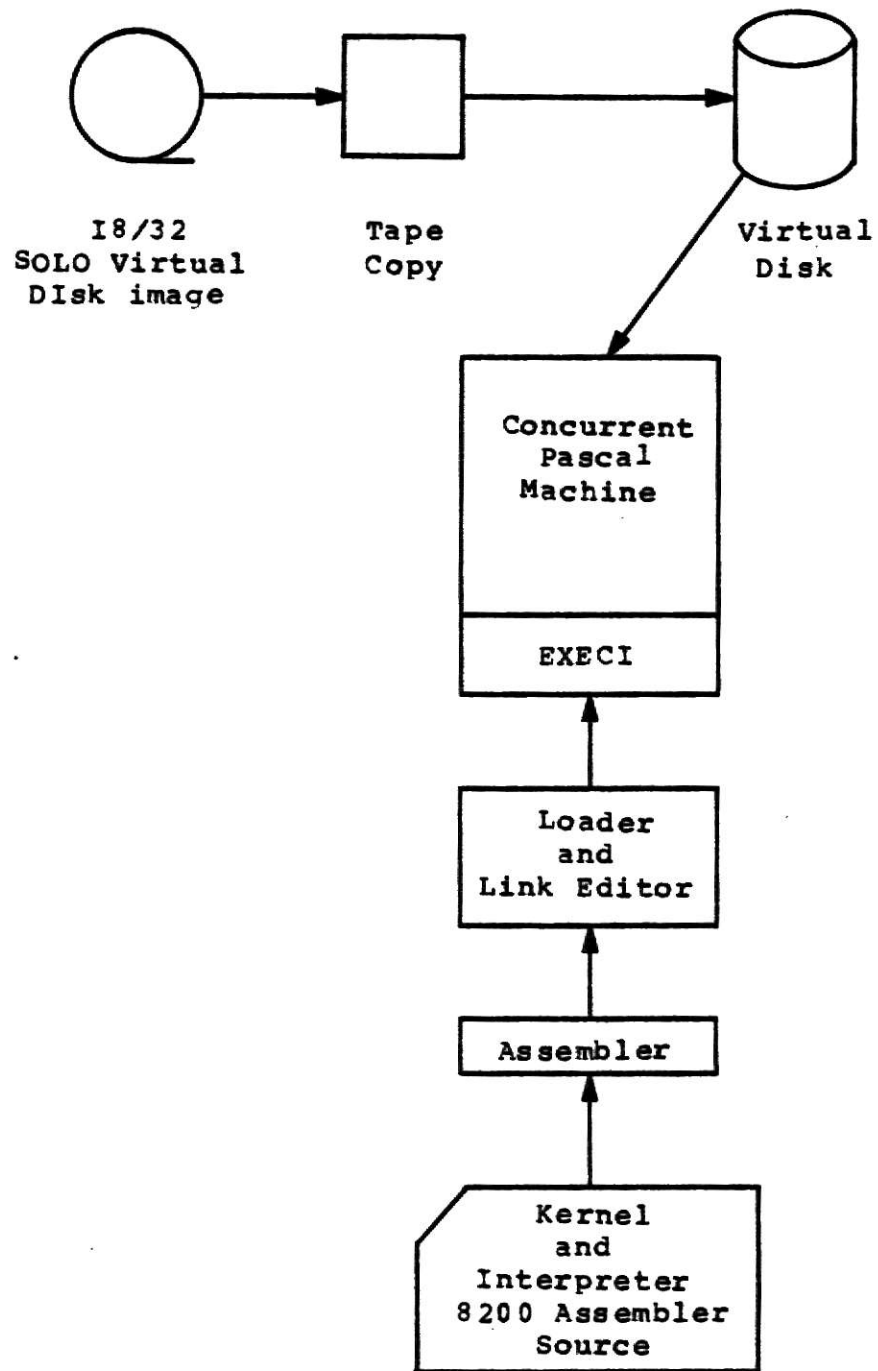


Figure 2.1 Concurrent Pascal Transportation

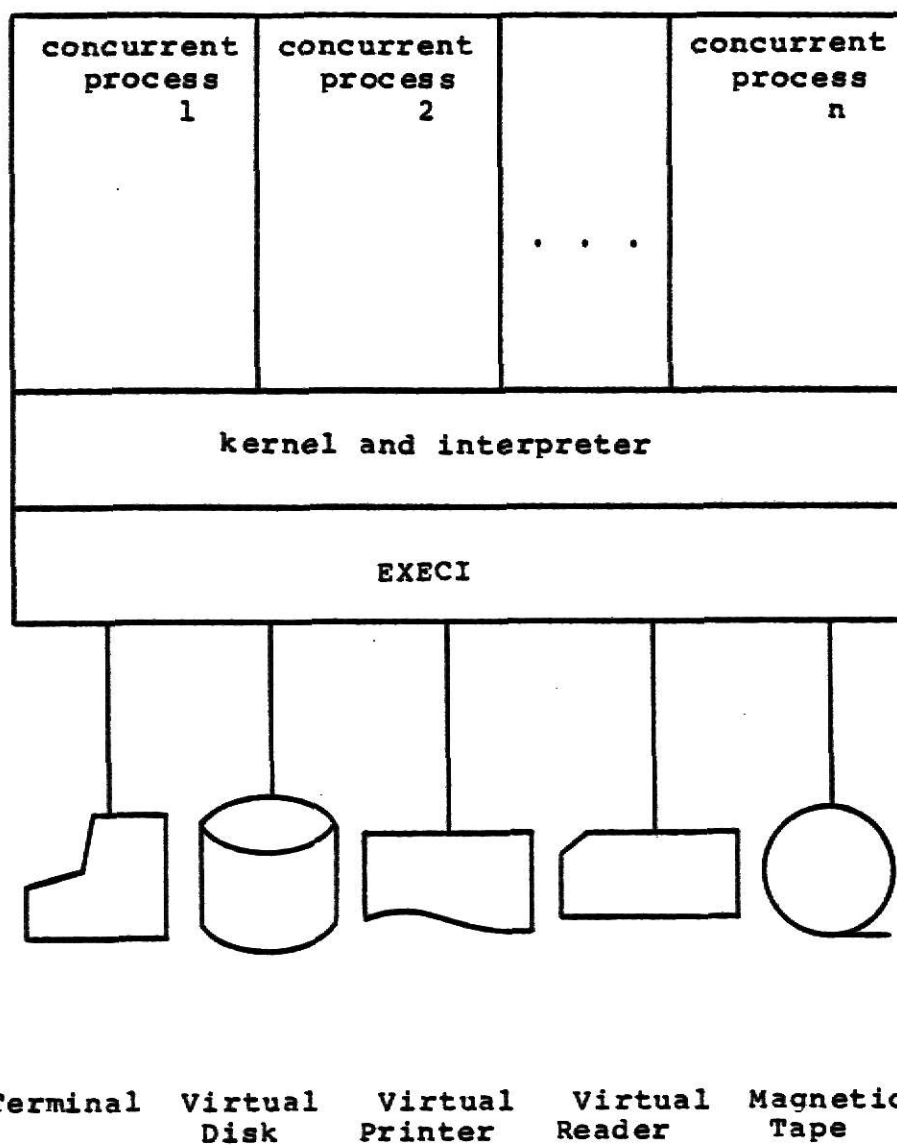


Figure 2.2 NCR8200 Concurrent Pascal Implementation

64K -->

EXECI
Kernel Interpreter
Monitor gates and Process records
free
Process n Private Data Space
.
Process 1 Private Data Space
Initial Process and Common Space

Figure 2.3 Memory Allocation

3. TRANSPORTING CONCURRENT PASCAL TO THE NCR8200

This chapter contains a discussion of the problems of porting Concurrent Pascal to the 8200. Since the 8200 is not a well known machine a few paragraphs are included that describe it. The problems of porting are in general distinct from the I8/32 implementation.

3.1 NCR8200 Hardware Architecture

The NCR8200 computer system is built around a 600 series processor[NCR2] which is used in a large number of NCR products.

The processor is a word addressable, general purpose register machine. The maximum addressable memory is 64K words. The memory is organized in a linear address space of 0 to 65535 in which each address identifies a sixteen bit word. Utilizing one and two word instructions seven permissible hardware addressing modes are available:

- 1) relative-direct
- 2) relative-indirect
- 3) absolute
- 4) absolute-indirect
- 5) indexed
- 6) absolute-indirect, post-indexed
- 7) immediate.

Instruction operands are register-memory or register-register. A memory-memory instruction is not available, a fact that impacts interpreter performance.

The processor has sixteen registers. Eight registers are used for IO, subroutine linkage, and machine state preservation during interrupts. The remaining eight

registers can be used for general programming, however, only four of these may be used as index registers.

Two types of IO are possible with the 600 series processors--direct memory access(DMA), which is similiar to most machines and transfers data directly between memory and high-speed devices, and auto data transfer (ADT), which is used to transfer data to slow-speed devices. ADT devices may require special data formats and interrupt service routine linkage must be maintained at the end of the data buffer as shown in figure 3.1.

3.2 Problems of Porting

The following sections contain a discussion of problems of the implementation that are distinct from the Interdata 8/32 implementation. Many of the problems encountered by the I8/32 implementators were not present in the NCR8200 implementation. Of the problems encountered, the largest was the segmentation addressing inherited from the PDP/11 implementation. The problem exists because the SOLO operating system is 72K bytes in size and data addresses are limited to sixteen bits. The PDP/11 implementation solved this with hardware segmentation registers which gave each Concurrent Pascal process a 64K byte address space, part of which was common with all other processes. The 8200 implementation solved the problem in a fashion similiar to the I8/32 implementation as will be discussed in relation to the byte address simulation.

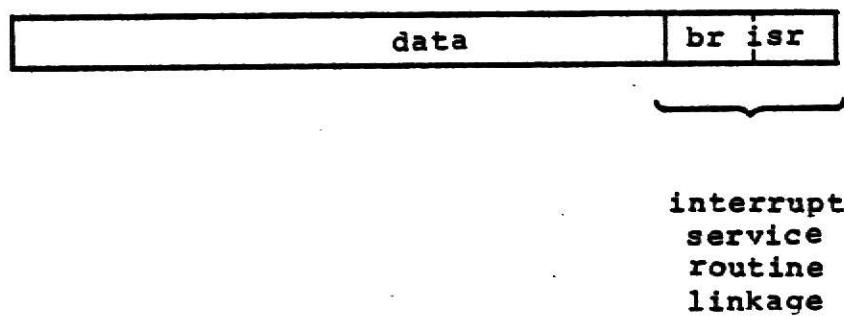


Figure 3.1 ADT IO Buffer

The problems to be discussed in the following sections are byte address simulation in conjunction with address segmentation, the lack of floating-point and integer multiply/divide hardware, and hardware requirements of IO buffers.

3.2.1 Byte Address Simulation and Address Mapping

A problem even more severe than the problem of address segmentation was the mapping of byte addresses onto a machine with a word addressable architecture. Even though the byte-to-word and word-to-byte conversions could be done with a single shift instruction the impact on efficiency is obvious. Each time a data address is pushed onto the stack it must be converted to a byte address and each time an address on the stack is used to access memory it must be converted back to a word address. This conversion overhead must also be added to the overhead of the software address segmentation.

Figure 3.2 illustrates the operations involved in using a data address that was removed from the stack to access memory. A sixteen bit byte address is converted to a word address. This address is then compared to the common top. If the word address is greater than common top it is an address of data in the private segment of a process. Therefore the displacement from the private segment base must be added to form the real memory address. If the address is less than or equal to the common top it is an address in the

common segment.

The reverse of the above operation, pushing a data address onto the stack is illustrated in figure 3.3. In this case the absolute word address is mapped to the logical address by subtracting the displacement off the private segment base if it is greater than the common top. The mapped address is then converted to a byte address that can be placed on the stack.

With performance in mind only data addresses are mapped and pushed on the stack as byte addresses. Linkage information, specifically the stack pointer, the global base, the local base, and the virtual code pointer are maintained as word addresses and are pushed on the stack as unmapped word addresses.

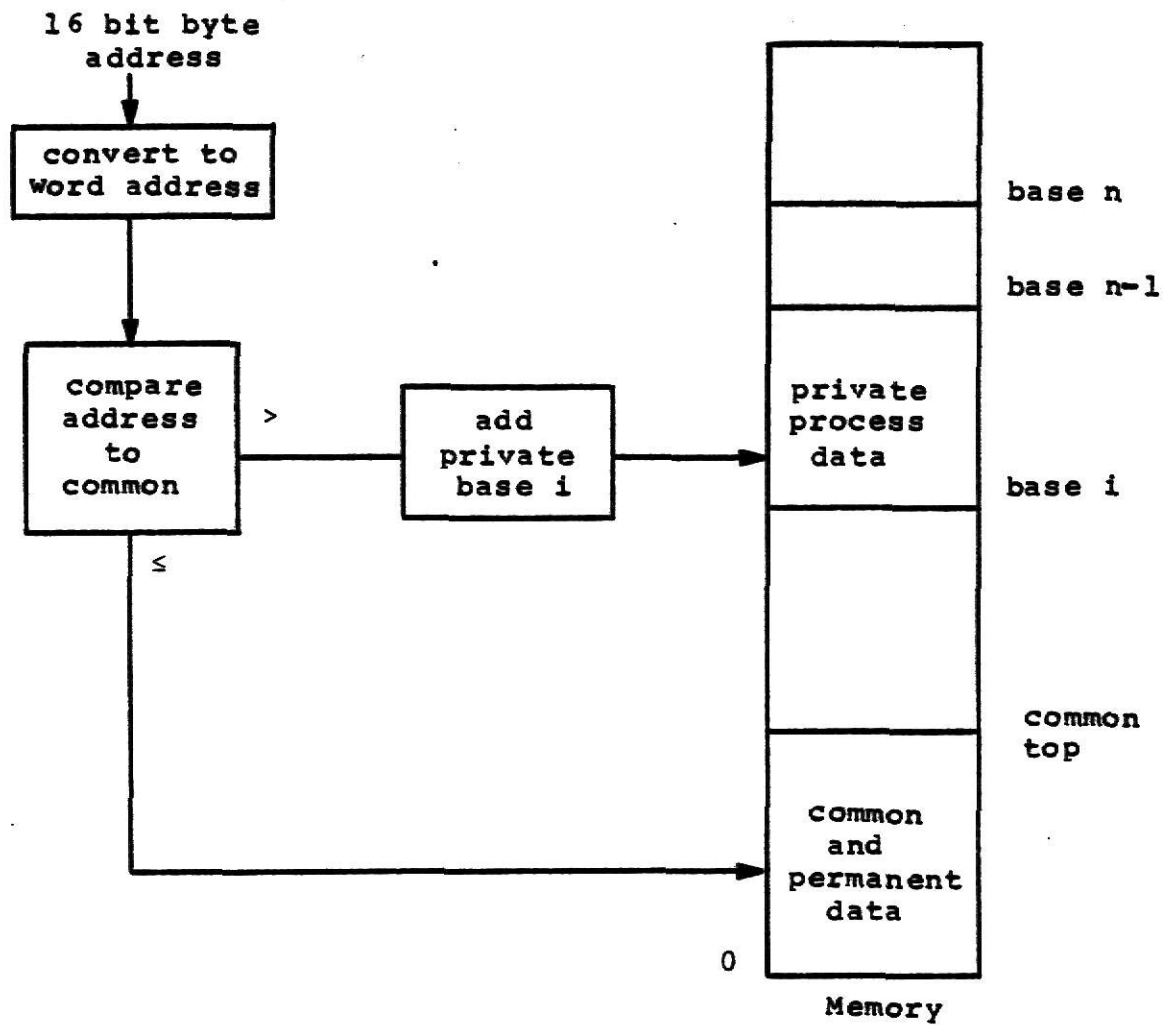


Figure 3.2 Byte Address Simulation and Address Segmentation

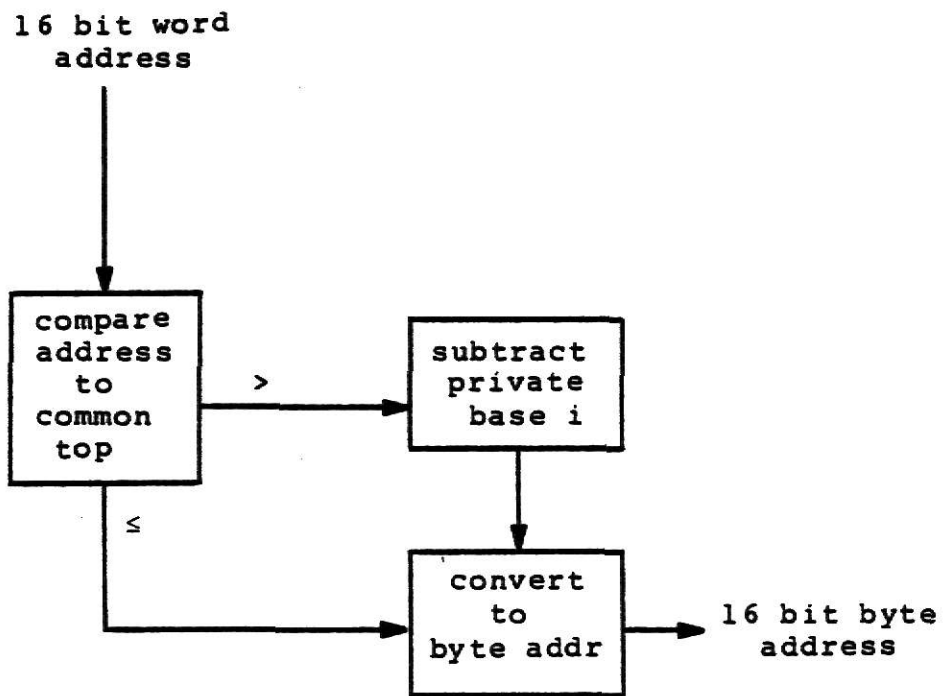


Figure 3.3 Pushing Data Addresses

3.2.2 IO Buffers

The hardware requirement of two words of interrupt linkage following each ADT data buffer prevents IO from being performed directly in the user's program space. Fortunately only the slower devices that use the ADT type IO require this expanded buffer. IO is performed in the kernel's address space and the data is copied to or from the user's program space.

3.2.3 Floating-point Emulation

A minor problem was the absence of floating-point and integer multiply/divide hardware on the 8200. Although this presented no conceptually difficult problems, a considerable amount of time had to be invested in developing software to make up for hardware deficiencies. While integer multiply/routines were readily adapted from existing software, the floating-point routines had to be implemented from scratch. Only the short form floating-point similar to the IBM/360 form was implemented. The accuracy of these routines is at best minimal but no problems have been found with the compilers of other SOLO utilities that use real numbers.

3.3 Portability Evaluation

The implementation of Concurrent Pascal on an 8200 is significant from a portability viewpoint. The first implementation was on a machine with byte addressable memory and hardware support for simulating a stack architecture.

Stack operations on the PDP/11 are facilitated by the use of autodecrementing and autoincrementing register modes[DEC]. The virtual code of the PDP/11 implementation with only the minor modifications made by the I8/32 implementors was transported to a machine with word addressable memory and no support for the stack architecture. Although accurate performance statistics are not available the system has performed sufficiently for prototype development work.

The resulting system and the manpower investment required to implement it speak well of the portability of the system. The system was implemented in approximately three man-months. The porting process was aided significantly by the availability of documentation. The logic of the kernel was documented by the Pascal like pseudo-code included as comments in the PDP/11 kernel and the intermediate level documentation of the I8/32 kernel. The I8/32 documentation included a module level functional description and the description of the module interface conventions. A very high level description of the Concurrent Pascal machine was provided by [BH3]. The 'Concurrent Pascal Implementation Notes'[BH4] provided supplemental information to the interpreter and kernel descriptions. The availability of a wide range of documentation allowed the non-input/output portion of the kernel to be coded in less than a week. Even more significant is that after correcting a few clerical errors

during early development the system has been used several months without detecting any errors attributed to the kernel.

The worst implementation problem was the byte address simulation in the interpreter. Converting data addresses between bytes and words and maintaining the linkage information in words caused many confusing moments in coding and debugging the interpreter. These problems have been eliminated by adapting the Pascal virtual code to the word addressable architecture of the 8200.

The largest manpower investment was in the IO portion of the kernel. Even though the decision was made to use existing drivers, the heterogeneous interface requirements of the drivers resulted in a significant amount of work.

4. ADAPTING THE CONCURRENT PASCAL MACHINE TO THE NCR8200

The most noted architectural difference between Brinch Hansen's Concurrent Pascal machine and the 8200 is the difference in addressing. This chapter contains a description of the addressing requirements of the data manipulation instructions in the byte addressable Pascal machine. Furthermore, the modifications that were made to allow more efficient interpretation of the virtual code by the 8200 are presented. The modifications redefine the Concurrent Pascal machine from a byte addressable machine into a word addressable machine.

4.1 Addressing

The PDP/11 implementation is based on a sixteen bit machine word for storage of addresses and data. An exception in the case of character strings allows the addressing of an individual character. Table 4.1 summarizes the storage requirements for the Pascal data types. The interpreter for the PDP/11 uses word level addressing instructions for manipulating integer, real, set, and character data types. Byte instructions are used only for manipulating individual characters in an array of characters. The only virtual instructions which access data on the byte level are COPYBYTE and PUSHBYTE.

The low order bit of the sixteen bit address is only significant for operations on character arrays. All other operations are performed on data that is aligned on word

boundaries. This alignment is required by the PDP/11 architecture[DEC]. A program exception occurs if a word operation is attempted on data located at an odd byte address.

The byte addressing Pascal machine manipulates character arrays as described below. Two basic operations are performed on a stack machine, pushing data onto the stack and copying data from the stack.

Pushing an element of a character array onto the stack is illustrated in figure 4.1. The source array address is pushed onto the stack with the LOCALADDR or GLOBALADDR virtual instruction. The subscript value is then pushed onto the stack by a PUSHLOCAL or PUSHGLOBAL or left on the stack as the result of an expression evaluation. The INDEX virtual instruction then calculates the source byte address by multiplying the subscript value times the array component length-- one in the case of character arrays--and adds the result to the source array address. The source byte address is left on the stack and used by the PUSHBYTE instruction to address the character array element. During the source byte address calculation the INDEX instruction validates the subscript value.

Common Pascal Data Types

Integer	1 word
Real	2 words
Set	8 words
Char	1 word
Array[1..n] of Char	n/2 words
(n must be even)	

Table 4.1 Data Storage Requirements

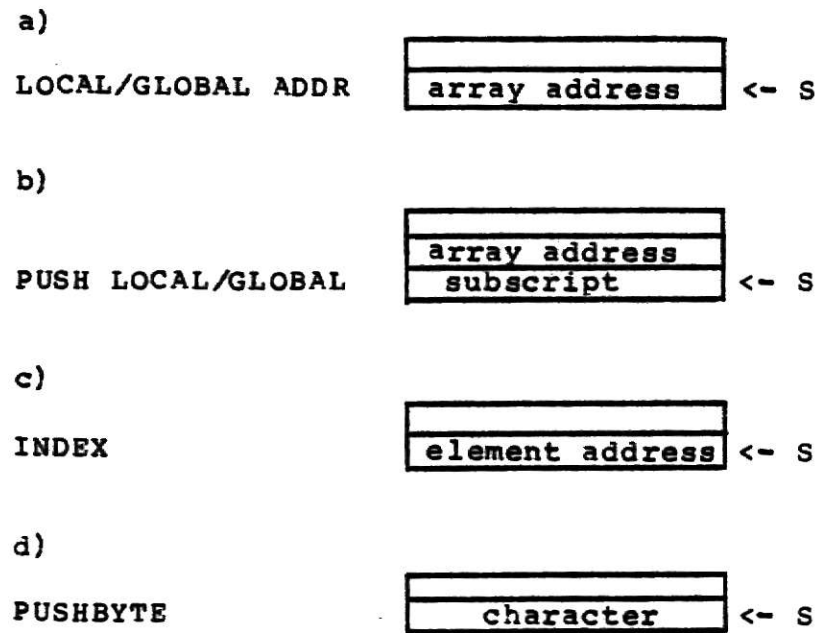


Figure 4.1 Push Character Array Element
(byte addressable)

Copying a character to a character array element is illustrated in figure 4.2. This operation is similar to the push operation with the INDEX instruction calculating the destination byte address. Figure 4.3 illustrates the virtual code for a Pascal statement that copies an element of a character array to a character variable.

The virtual code and stack operations are shown in figure 4.4 for copying an element of a character array to a character variable. S is the stack pointer and B is the local base pointer. The number on the virtual instruction corresponds to the picture of the stack taken after execution of the instruction. The numbers on the left of the stack pictures are arbitrary memory addresses and the labels on the right are variable names.

a)

LOCAL/GLOBAL ADDR array address <- S

b)

PUSH LOCAL/GLOBAL array address
 subscript <- S

c)

INDEX element address <- S

d)

PUSH LOCAL/GLOBAL element address
 character <- S

e)

COPYBYTE <- S

Figure 4.2 Copy Character to Array Element
(byte addressable)

Pascal statements

```

.
.
.
var text : array[1..6] of char;
var c:char;
.
.
.
text:='pascal';
.
.
.
c:= text[2];
.
.
.

```

Virtual Code for c:=text[2];

	Instruction	Arguments	
1	LOCALADDR	-4	"c"
2	LOCALADDR	-10	"text"
3	PUSHCONST	2	
4	INDEX	1,5,1	
5	PUSHBYTE		
6	COPYBYTE		

Figure 4.3 Virtual Code for Character Array Operations
(byte addressable)

After execution of instructions
1, 2, and 3 the stack appears
as follows:

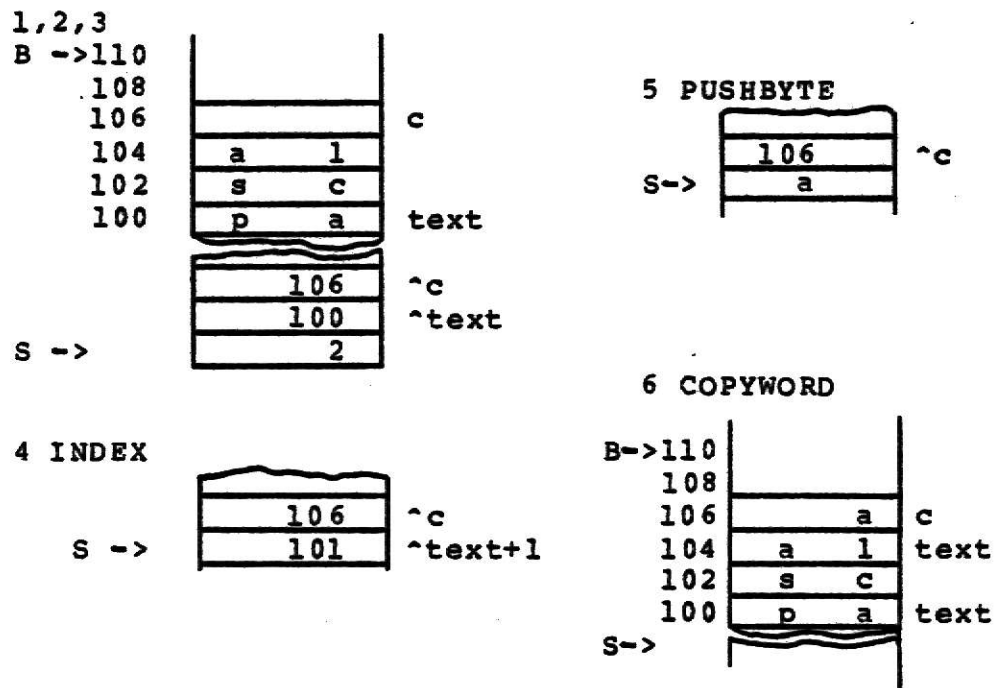


Figure 4.4 Stack Pictures of Array Element Copy
(byte addressable)

4.2 Virtual code modification

The adaptation of the virtual code for more efficient addressing was a relatively simple procedure once the first implementation was running and the compilers were available. A pass was added to the existing sequential and concurrent compilers to effect the necessary code modifications. Then SOLO and its utilities were recompiled. Once this was completed the necessary interpreter changes were made and the new system was working.

Figure 4.5 shows how the code in figure 4.3 appears after modification. Any instruction arguments which were not previously specified in words were divided by two. This had no impact on the instructions which operated at the word level. Only the COPYBYTE, PUSHBYTE, and INDEX instructions had to be interpreted differently to retain byte level accessibility for character arrays. The psuedo-code for the interpreter is included in the Appendix.

Figures 4.6 and 4.7 show how pushing and copying of an element of a character array is implemented with word level addressing. The length argument for the INDEX instruction is now the length of the array element in words. In the case of character arrays this argument is zero. If the argument is zero the INDEX instruction does not calculate the element address, instead it calculates the zero relative offset of the character element and performs the necessary subscript range checks. This results in the array address

and the validated byte offset of the desired array element on the top of the stack. The interpretation of the PUSHBYTE and COPYBYTE instructions has been changed to perform the array element address calculation. The virtual code and stack pictures for copying a character array element to a character variable (figure 4.3) with word level addressing is shown in figure 4.8.

A minor problem was encountered in changing the interpretation of the COPYBYTE and INDEX instructions. The copying operation pushes one more word on the stack than the byte level addressing implementation. This extra word may cause the temporary stack requirements for a procedure to be one word greater than that calculated by the compiler. The temporary variable space is allocated by the procedure entry instructions. The more aesthetically pleasing solution would be to change the compiler to allow for this extra word. However, due to the amount of time available for this project the interpreter allows one more word than asked for by the procedure entry instructions.

Virtual Code	Instructions	Arguments	
1	LOCALADDR	-2	"c"
2	LOCALADDR	-5	"text"
3	PUSHCONST	2	
4	INDEX	1,5,0	
5	PUSHBYTE		
6	COPYWORD		

Figure 4.5 Virtual Code for Character Array Operations
(word addressable)

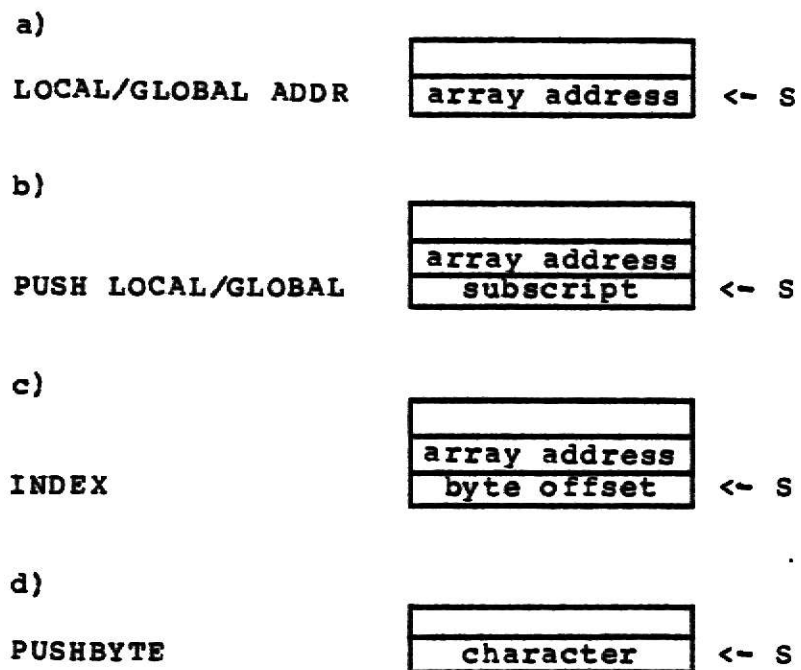


Figure 4.6 Push Character Array Element
(word addressable)

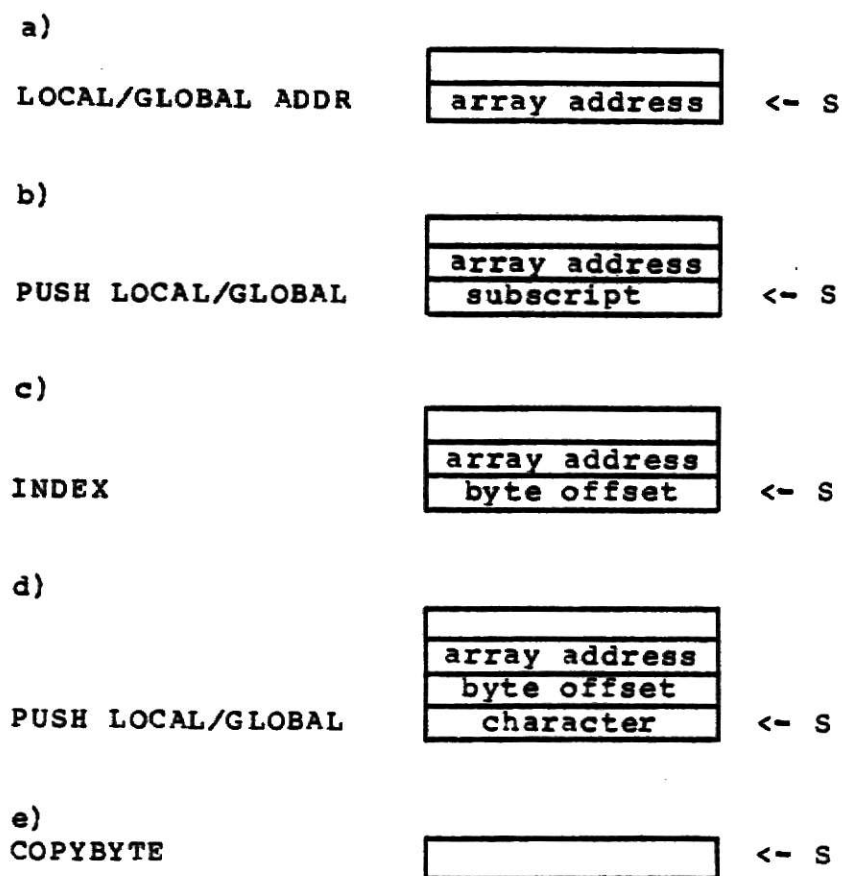
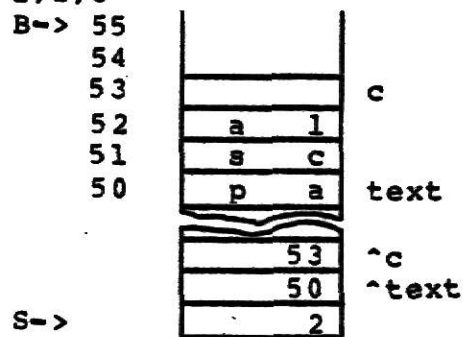


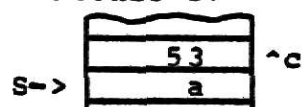
Figure 4.7 Copy Character to Array Element
(word addressable)

After execution of instructions
1, 2, and 3 the stack appears
as follows:

1,2,3



5 PUSHBYTE.



6 COPYWORD

4 INDEX

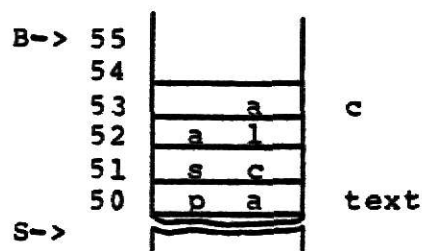
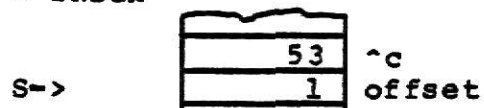


Figure 4.8 Stack Pictures of Array Element Copy
(word Addressable)

4.3 Adaptability Evaluation

The changes necessary to adapt the virtual code to a word addressable architecture were suprisingly simple. It is doubtful that Hansen designed the system to be adapted in this fashion so one might conclude that the high adaptability is a result of the high portability.

The change to word level addressing in the virtual code simplified the interpreter by allowing the removal of the shift instructions which implemented the word-to-byte and byte-to-word address conversions. An even greater improvement was the elimination of the need for software address segmentation. Since the entire memory of the 8200 can be addressed with a sixteen bit address, which can be pushed onto the stack, there is no requirement for segmentation. The removal of the address conversion code and the software segmentation code has resulted in an approxmate fifteen percent performance improvement.

5. A GENERAL IO ARCHITECTURE FOR THE PASCAL KERNEL

This chapter describes a general IO architecture for the Pascal kernel. A general architecture was necessary for the development of the system primarily because of the difficulty of implementing drivers for the 8200. In determining the implementation approach for the system it was decided that as many existing drivers as possible should be utilized. This decision introduced the problem of interfacing the Pascal devices to existing "off the shelf" drivers with heterogeneous interfaces and system requirements. A solution to this problem is the general IO architecture described in this chapter.

5.1 Driver Classification

After a survey of the available drivers they were grouped into the following three classifications based on the driver's method of handling IO initiation and completion:

- 1) Drivers which initiate the IO operation and set a status cell in a parameter block busy before returning control. The driver's interrupt service routine sets the status cell with the completed status and restores the interrupted state.
- 2) Drivers which initiate the IO operation and return control. The driver must be polled for the device's completion status. The polling is often a requirement of device design.

3) Drivers which initiate the IO operation and return control. The interrupt service routine schedules a user defined completion routine after handling device dependent functions.

5.2 Mapping Pascal IO to Existing Devices

An interface routine was written for each driver to map the Pascal IO device to the driver. Figure 5.1 illustrates the levels an IO request passes through. The driver interface routines map the requirements of the Pascal devices to the "off the shelf" driver for which they were written. In most cases these routines are only a few lines of code that map the Pascal IOPARAM block to the parameter block for the driver and the translation of completion statuses into Pascal statuses.

The IO request class in the kernel is driven by the IO device structures illustrated in figure 5.2. The Pascal virtual device table contains an entry for each device in the order in which they were enumerated in the concurrent program. Each of these entries point to a device control block which contains information about that particular device type. The device control block contains the addresses of the open routine which initializes the device when the system is loaded, the request initiation routine, the device profile which describes some attributes of the device, and addresses of one or more unit control blocks. The device profile indicates to the system configuration

procedures if buffers are to be allocated in the kernel's address space for ADT type IO and if the device must be polled. Only one device control block exists for each device type. A unit control block(UCB), of which there is one corresponding to each unit of a device type, contains driver dependent information. The only system information is the status and the user, the concurrent Pascal process that is using the driver. The use of a UCB for each device unit enables a driver to control more than a single unit.

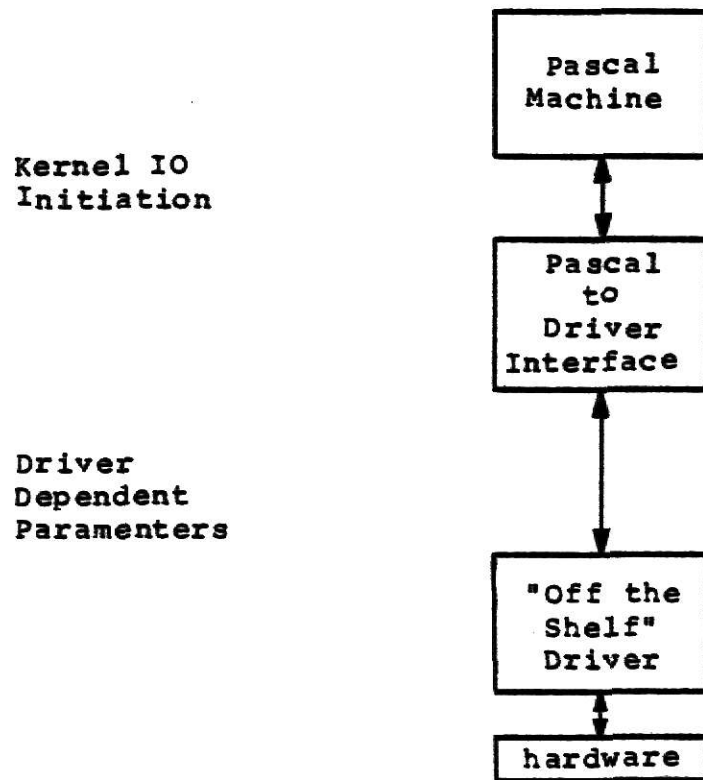


Figure 5.1 IO Levels(Pascal to Hardware)

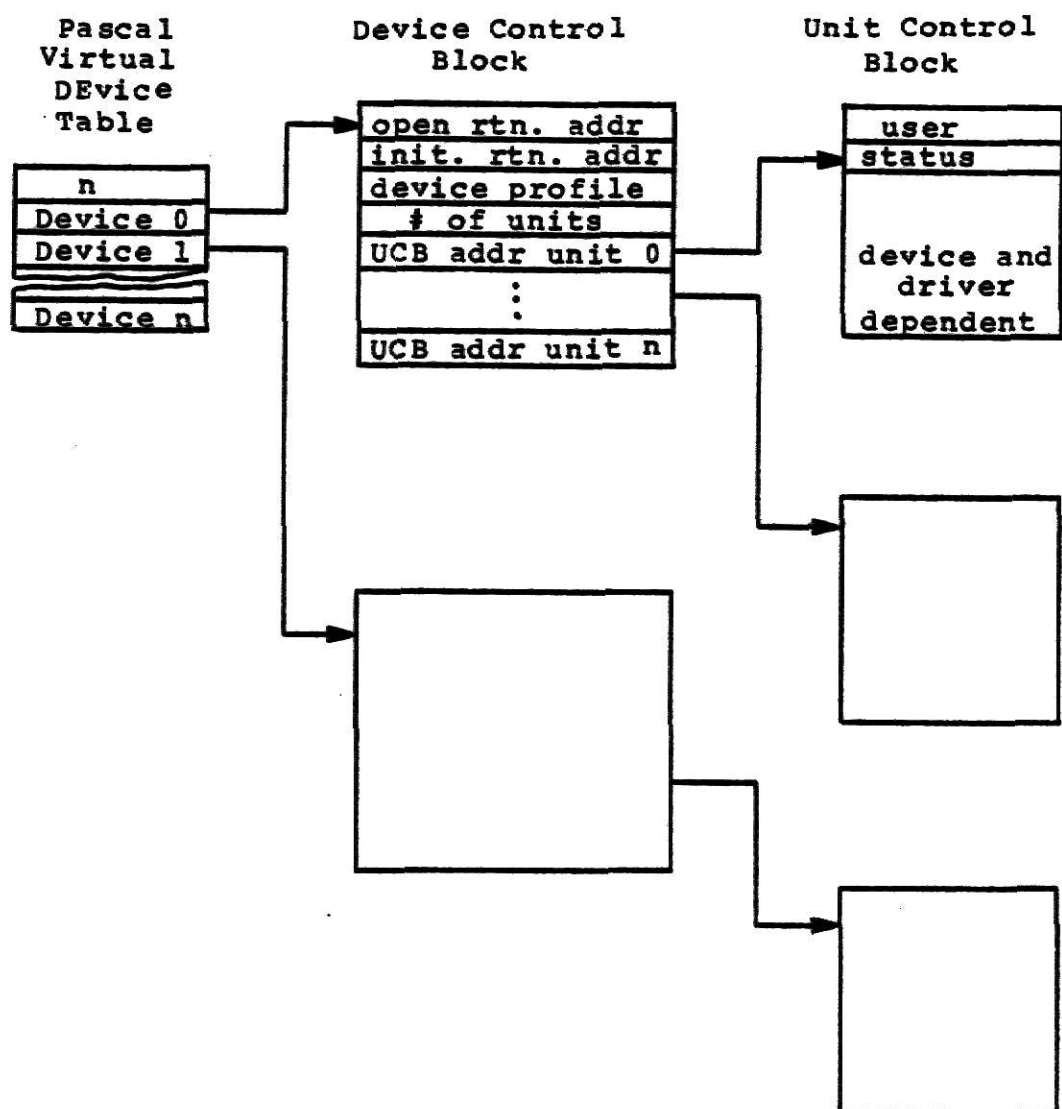


Figure 5.2 IO Device Structures

5.3 IO Request Handling

The initiation, interrupt handling, and completion of an IO request are diagrammed in figure 5.3. After the IO activity is initiated class one(1) and two(2) driver requests are placed on the poll queue. The kernel polling routines, which are driven by an interval timer, administer the poll queue. For class one drivers the kernel polling routines check the UCB status cell and schedule the completion routines when the status is complete. For class two drivers the kernel polling routines call the driver's polling routine, which tests and returns the device status. The time interval on the poll queue is determined by the device's speed. The scheduling of the completion routine for class three(3) drivers is performed by the device interrupt service routine upon an device interrupt.

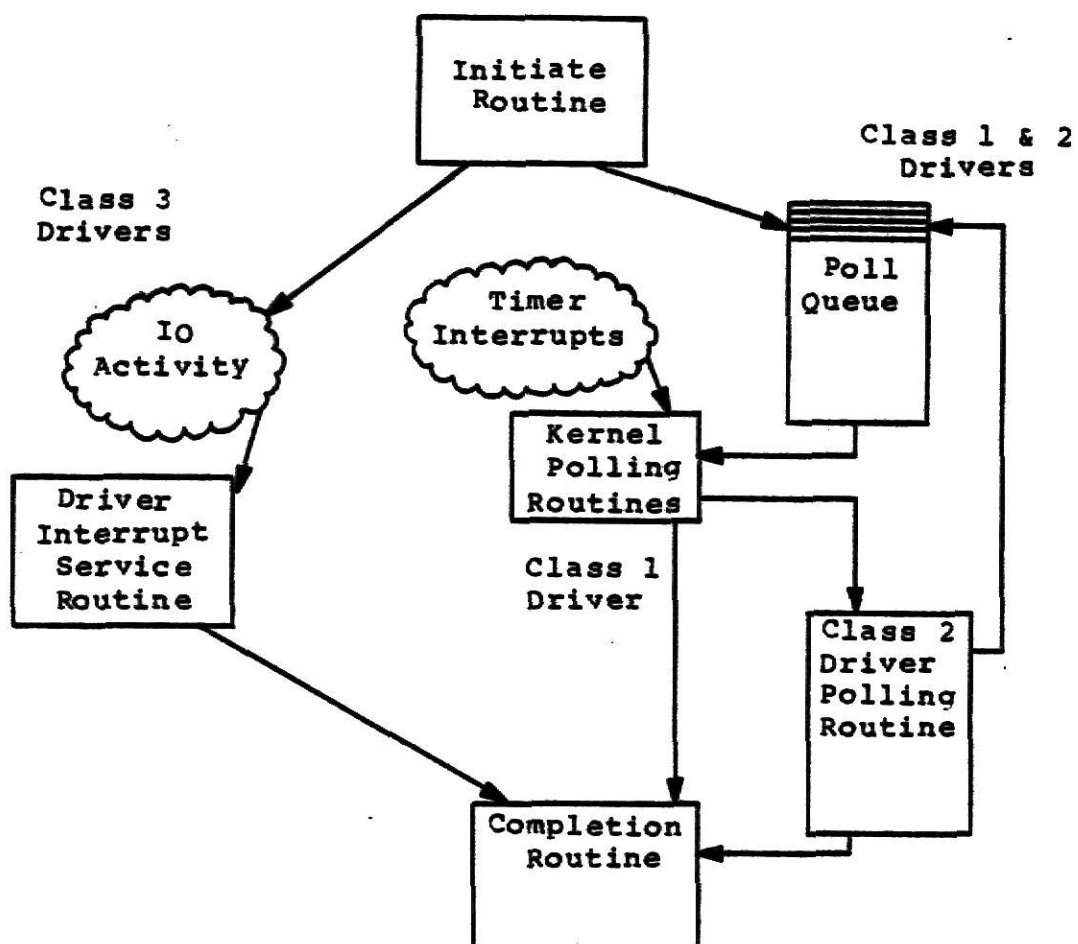


Figure 5.3 IO Request Handling

6. SUMMARY

Two implementations of Concurrent Pascal on the NCR8200 are described in this report. The first was a pure porting of Concurrent Pascal to the 8200 With no modifications to the virtual code. The second implementation involved adapting the virtual code to the word addressable architecture of the 8200. The only significant difference between the two implementations is the interpreter. The first interpreter was the most difficult to implement as it had to deal with byte address simulation and address segmentation. The second implementation of the interpreter is similiar to the PDP11 interpreter and easy to understand. This simpler implementation resulted in an approximate fifteen percent performance improvement over the first implementation. The implementation on the 8200, which is architecturally a much different machine than the PDP11, illustrates the high portability and high adaptability of the Concurrent Pascal machine.

The general IO architecture of the Pascal kernel has been a successful undertaking. Two drivers have been added to the system with less than two days of effort for each. One of these, which is a diagnostic driver for a new peripheral, allowed the use of the device in a version of SOLO before the device was functioning.

Table 6.0 summarizes the development effort for the 8200 system. The man hours column reflects the importance

of good documentation to a system implementation. The IO Kernel was the only portion of the implementation that required extensive design, and it required the largest amount of time per line of code. Time was not available to undertake an extensive system test. However, the system has been used for several months to make changes to SOLO and develop several Sequential Pascal programs without uncovering any errors.

The implementation of the Concurrent Pascal system was a rewarding and informative experience. The system has proven valuable for prototype development work on a Data Base Management System by providing, at a low man-power cost, a high-level systems language on a machine which previously had none.

	Lines of New Code	Lines of Existing Code	Estimated Man Hours(*)
Interpreter	1400		160
Floating-Point Routines	375		40
Integer Math Routines		50	
Kernel	1150		40
IO Kernel	200		80
Interface Routines	662		100
Drivers		1640	
	<hr/>	<hr/>	<hr/>
Totals	3778	1690	420

(*) Number includes coding and debug time.

Table 6.0 Summary of Development Effort

References

- [BH1] Brinch Hansen, P., The Programming Language Concurrent Pascal. IEEE Transactions on Software Engineering, Vol. 1, No. 2 (June 1975), pp. 199-207.
- [BH2] ---, The Architecture of Concurrent Programming. Prentice-Hall Inc., Englewood Cliffs, NJ, 1977.
- [BH3] ---, Concurrent Pascal Implementation Notes. Information science, California Institute of Technology, January 1976.
- [DEC] Digital Equipment Corp. PDP11/45 Processor Handbook, 1973.
- [DN1] Neal, David, An Architectural Base for Concurrent Pascal. (M.S. Thesis). Kansas State University Department of Computer Science, Technical report CS 76-19, January, 1977.
- [DN2] Neal, David, et. al., KSU Implementation of Concurrent Pascal-- A Reference Manual. Kansas State University Department of Computer Science, Technical report CS 76-16, January, 1977.
- [NCR1] NCR Corp. Programmer's Reference Manual--NCR605 EXECI Operating System. 1975.
- [NCR2] ---, NCR605 Reference Manual. Stock Number ST-9417-01, 1975.
- [PW] Poole, P. C., and Waite, N. M., Portability and

Adaptability. in Bauer, F. L., Advanced Course on
Software Engineering, Springer-Verlag, 1973.

Appendix

Pseudo-Code of Concurrent Pascal Interpreter
for a Word Addressable Architecture

The pseudo-code procedures in this appendix define the virtual instructions for the Concurrent Pascal Machine. These procedures were copied from the interpreter for the PDP11 system and modified for a word addressable architecture. The notation $s:+1$ means $s:=s+1$ and $st(s)$ means $store(s)$.

```

" s    - stack pointer
  g    - global base
  b    - local base
  q    - virtual code pointer
"
procedure constaddr(displ);
begin
  test st(job);
  if zero
    then s:-1; st(s):=st(constaddr) "system"
    else s:-1; st(s):=st(g+5);      "job"
  st(s):+st(q); q:+1;
end;

procedure localaddr(displ);
begin
  s:-1; st(s):=b;
  st(s):+st(q); q:+1;
end;

procedure globaladdr(displ);
begin
  s:-1; st(s):=g;
  st(s):+st(q); q:+1;
end;

procedure pushconst(value);
begin
  s:-1; st(s):=st(q); q:+1;
end;

procedure pushlocal(displ);
begin
  w:=b;
  w:+st(q); q:+1;
  s:-1; st(s):=st(w);

```

[illegible]

```

begin
    st(s):+st(q); q:+1;
end;

procedure index(min,max-min,length);
begin
    x:=st(s);
    x:-st(q); q:+1;
    if less then
        goto rangeerror;
    x compare st(q); q:+1;
    if greater then
        goto rangeerror;
    test st(q); q:+1;
    if not zero then
        begin
            x:*st(q-1);
            s:+1;
            st(s):+x;
        end
    else
        st(s):=x;
    end;
end;

procedure pointer;
begin
    test st(s);
    if zero then
        goto pointererror;
end;

procedure variant(displ,tagset)
begin
    w:=1;
    x:=st(s);           "x=record addr"
    x:+st(q); q:+1;     "x=tag addr"
    w: shift st(x);     "w=1 shift tagvalue"
    st(q) testbit w; q:+1;
    if bitzero then
        goto variantererror;
end;

procedure range(min,max);
begin
    st(s) compare st(q); q:+1;
    if less then
        goto rangeerror;
    st(s) compare st(q); q:+1;
    if greater then
        goto rangeerror;
end;

```

```
procedure copybyte;
```

```
begin
  w:=st(s+1);
  wx: shift 1;
  w:=st(s+1);
  z:=st(w);
  y:=st(s);
  x bittest &8000;
  if bitzero then
    begin
      y: shift 8;
      z: and    &00ff;
    end
  else
    z: and    &ff00;
    z: or    y;
    st(w):=z;
    s:=s+3;
  end;
```

```
procedure copyword;
```

```
begin
  st(st(s+1)):=st(s); s:=s+2;
end;
```

```
procedure copyreal;
```

```
begin
  w:=st(s+2);
  st(w):=st(s); w:=w+1; s:=s+1;
  st(w):=st(s); w:=w+1; s:=s+1;
  test st(s); s:=s+1;
end;
```

```
procedure copyset;
```

```
begin
  w:=st(s+8);
  st(w):=st(s); w:=w+1; s:=s+1;
  st(w):=st(s); w:=w+1; s:=s+1;
  st(w):=st(s); w:=w+1; s:=s+1;
  st(w):=st(s); w:=w+1; s:=s+1;
  st(w):=st(s); w:=w+1; s:=s+1;
  st(w):=st(s); w:=w+1; s:=s+1;
  st(w):=st(s); w:=w+1; s:=s+1;
  test st(s); s:=s+1;
end;
```

```
procedure copytag(length);
```

```
begin
  st(st(s+1)):=st(s); s:=s+1;
  w:=st(q); q:=q+1;
  x:=st(s); s:=s+1;
```

```
"length>0"
```

```
"w=length"
```

```
"x=tag addr"
```

```

    test st(x); x:=+1;
    iterate w times
        clear st(x); x:=+1;
end;

```

```

procedure copystruc(length);
begin
    w:=st(q); q:=+1;           "w=length"
    x:=st(s); s:=+1;           "x=source addr"
    y:=st(s); s:=+1;           "y=dest addr"
    iterate w times
        st(y):=st(x); y:=+1; x:=+1;
end;

```

```

procedure new(stacklength+length,length);
begin
    x:=b;
    x:=st(heaptop);
    x compare st(q); q:=+1;
    if less<unsigned> then
        goto heaplimit;
    st(st(s)):=st(heaptop); s:=+1;
    st(heaptop):+st(q); q:=+1;
end;

```

```

procedure newinit(stacklength+length,length);
begin
    "length>0"
    x:=b
    x:=st(heaptop);
    x compare st(q); q:=+1;
    if less<unsigned> then
        goto heaplimit;
    st(st(s)):=st(heaptop); s:=+1;
    w:=st(q); q:=+1;
    st(heaptop):+w;
    x:=st(heaptop);
    iterate w times
        x:=+1; clear st(x);
end;

```

```

procedure not;
begin
    st(s):=-st(s);
    increment st(s);
end;

```

```

procedure andword;
begin
    w:=st(s); s:=+1;
    w:=not w;
    st(s):andnot w;
end;

```

```

procedure andset;
begin
  w:=8;
  iterate w times
  begin
    st(s):=not st(s);
    st(s+8):andnot st(s); s:+1;
  end;

end;

```

```

procedure orword;
begin
  st(s+1):or st(s); s:+1;
end;

```

```

procedure orset;
begin
  w:=8;
  iterate w times
    st(s+8):or st(s); s:+1;
end;

```

```

procedure negword;
begin
  st(s):=-st(s)
  if overflow then
    goto overflowerror;
end;

```

```

procedure negreal;
begin
  st(s):=<real>-st(s);
end;

```

```

procedure addword;
begin
  st(s+1):+st(s); s:+1;
  if overflow then
    goto overflowerror;
end;

```

```

procedure addreal;
begin
  w:<real>st(s); s:+2;
  w:++<real>st(s);
  st(s):=<real>w;
end;

```

```

procedure subword;
begin

```



```

    st(s+1):=-st(s);  s:=+1;
    if overflow then
        goto overflowerror;
    end;

procedure subreal;
begin
    w:=<real>st(s); s:=+2;
    x:=<real>st(s);
    x:=<real>w;
    st(s):=<real>x;
end;

procedure subset;
begin
    w:=2;
    iterate w times
        st(s+8):=andnot st(s); s:=+1;
    end;

procedure mulword;
begin
    x:=st(s); s:=+1;
    carry:=false;
    x:=st(s);
    if carry then
        goto overflowerror;
    st(s):=x;
end;

procedure mulreal;
begin
    w:=<real>st(s); s:=+2;
    w:=<real>st(s);
    st(s):=<real>w;
end;

procedure divword;
begin
    x:=st(s+1);
    extendsign w;
    wx:=st(s); s:=+1;
    if overflow then
        goto overflowerror;
    st(s):=w;
end;

procedure divreal;
begin
    w:=<real>st(s+2);
    w:=<real>st(s); s:=+2;
    st(s):=<real>w;
end;

```

```

end;

procedure modword;
begin
  x:=st(s+1);
  extendsign w;
  wx:/st(s); s:+1;
  if overflow then
    goto overflowerror;
  st(s):=x;
end;

procedure buildset;
begin
  w:=st(s); s:+1;
  if w<0 then goto rangeerror;
  w compare 127;
  if greater then
    goto rangeerror;
  x:=w; "x=member"
  w:mod 16; "w=member mod 16"
  x:div 16;
  x:+s; "x=set byte adr"
  y:=1;
  y:shift w; "y=set byte bit"
  st(x):or y;
end;

procedure inset;
begin
  w:=st(s+8);
  if w<0 then goto rangeerror;
  w compare 127;
  if greater then
    goto rangeerror;
  x:=w; "x=member"
  w:mod 16; "w=member mod 16"
  x:div 16;
  x:+s; "x=set byte adr"
  y:= st(x); "y=set byte"
  w:=-w;
  y:shift w;
  y:mod 1; "y=set bit"
  s:+8;
  st(s):=y;
end;

procedure lsword;
begin
  clear w;
  st(s) compare st(s+1); s:+1;
  if greater then

```

```

        increment w;
        st(s):=w;
    end;

procedure eqword;
begin
    clear w;
    st(s) compare st(s+1); s:=+1;
    if equal then
        increment w;
    st(s):=w;
end;

procedure grword;
begin
    clear w;
    st(s) compare st(s+1); s:=+1;
    if less then
        increment w;
    st(s):=w;
end;

procedure nlword;
begin
    clear w;
    st(s) compare st(s+1); s:=+1;
    if notgreater then
        increment w;
    st(s):=w;
end;

procedure neword;
begin
    clear w;
    st(s) compare st(s+1); s:=+1;
    if notequal then
        increment w;
    st(s):=w;
end;

procedure ngword;
begin
    clear w;
    st(s) compare st(s+1); s:=+1;
    if notless then
        increment w;
    st(s):=w;
end;

procedure lsreal;
begin
    clear w;

```

```

    x:=<real>st(s); s:+2;
    st(s) compare x; s:+2;
    copyconditions;
    if less then
        increment w;
    s:-1; st(s):=w;
end;

```

```

procedure eqreal;
begin
    clear w;
    x:=<real>st(s); s:+2;
    st(s) compare x; s:+2;
    copyconditions;
    if equal then
        increment w;
    s:-1; st(s):=w;
end;

```

```

procedure grreal
begin
    clear w;
    x:=<real>st(s); s:+2;
    st(s) compare x; s:+2;
    copyconditions;
    if greater then
        increment w;
    s:-1; st(s):=w;
end;

```

```

procedure nlreal
begin
    clear w;
    x:=<real>st(s); s:+2;
    st(s) compare x; s:+2;
    copyconditions;
    if notless then
        increment w;
    s:-1; st(s):=w;
end;

```

```

procedure nereal;
begin
    clear w;
    x:=<real>st(s); s:+2;
    st(s) compare x; s:+2;
    copyconditions;
    if notequal then
        increment w;
    s:-1; st(s):=w;
end;

```

```

procedure ngreal
begin
  clear w;
  x:=<real>st(s); s:+2;
  st(s) compare x; s:+2;
  copyconditions;
  if notgreater then
    increment w;
  s:-1; st(s):=w;
end;

procedure eqset;
begin
  clear w;
  x:=s;
  y:=8;
  repeat
    st(x+8) compare st(x); x:+1;
    y:-1;
  until (y=0) or notequal;
  if equal then increment w;
  s:+15;
  st(s):=w;
end;

procedure nlset;
begin
  clear w;
  x:=s;
  y:=8;
  repeat
    st(x):andnot st(x+8); x:+1;
    y:-1;
  until (y=0) or notzero;
  if zero then increment w;
  s:+15;
  st(s):=w;
end;

procedure neset;
begin
  w:=1;
  x:=s;
  y:=8;
  repeat
    st(x+8) compare st(x); x:+1;
    y:-1;
  until (y=0) or notequal;
  if equal then clear w;
  s:+15;
  st(s):=w;
end;

```

```

procedure ngset;
begin
  clear w;
  x:=s;
  y:=8;
  repeat
    st(x+8):andnot st(x); x:+1;
    y:-1;
  until (y=0) or notzero;
  if zero then increment w;
  s:+15;
  st(s):=w;
end;

procedure lsstruct(length);
begin
  w:=st(q); q:+1;
  x:=st(s); s:+1;
  y:=st(s);
  clear st(s);
  repeat
    st(y) compare st(x); y:+1; x:+1;
    w:-1;
  until (w=0) or notequal;
  if less then
    increment st(s);
end;

procedure eqstruct(length);
begin
  w:=st(q); q:+1;
  x:=st(s); s:+1;
  y:=st(s);
  clear st(s);
  repeat
    st(y) compare st(x); y:+1; x:+1;
    w:-1;
  until (w=0) or notequal;
  if equal then increment st(s);
end;

procedure grstruct(length);
begin
  w:=st(q); q:+1;
  x:=st(s); s:+1;
  y:=st(s);
  clear st(s);
  repeat

```

```

    st(y) compare st(x); y:+1; x:+1;
    w:-1;
    until (w=0) or notequal;
    if greater then
        increment st(s);
    end;

```

```

procedure nlstruct(length);
begin
    w:=st(q); q:+1;
    x:=st(s); s:+1;
    y:=st(s);
    clear st(s);
    repeat
        st(y) compare st(x); y:+1; x:+1;
        w:-1;
    until (w=0) or notequal;
    if notless then
        increment st(s);
    end;

```

```

procedure nestruct(length);
begin
    w:=st(q); q:+1;
    x:=st(s) s:+1;
    y:=st(s);
    st(s):=1;
    repeat
        st(y) compare st(x); y:+1; x:+1;
        w:-1;
    until (w=0) or notequal;
    if equal then clear st(s);
    end;

```

```

procedure ngstruct(length);
begin
    w:=st(q); q:+1;
    x:=st(s); s:+1;
    y:=st(s);
    clear st(s);
    repeat
        st(y) compare st(x); y:+1; x:+1;
        w:-1;
    until (w=0) or notequal;
    if not greater then
        increment st(s);
    end;

```

```

procedure funcvalue(kind);

```

```

begin
  case kind of
    simpleword:                                "0"
      begin
        s:-1; clear st(s);
      end;
      "filler"
    simplereal:                                "4"
      begin
        s:-2;
      end;
      "filler"
    classword:                                "8"
      begin
        w:=st(s);
        clear st(s);
        s:-1; st(s):=w;
      end;
    classreal:                                "12"
      begin
        w:=st(s);
        s:-2;
        st(s):=w;
      end;
  end;
end;

procedure jump(distance);
begin
  q:+st(q);
end;

procedure falsejump(distance);
begin
  if (st(continue) = 0)
    &
    (st(job) <> 0)
  then goto exception
  else
  begin
    test st(s); s:+1;                                "continue=1"
    if zero
      then q:+st(q)
      else q:+1;
    end
  end;
end;

procedure casejump(min,max-min,distances);
begin
  w:=st(s); s:+1;
  w:-st(q); q:+1;
  if less then goto rangeerror;

```



```

    w compare st(q); q:+1;
    if greater then
        goto rangeerror;
    q:=w;
    q:=+st(q);
end;

```

```

procedure initvar(length);
begin
    w:=st(q); q:+1;
    x:=s;
    iterate w times
        clear st(x); x:+1;
end;

```

```

procedure call(distance);
begin
    w:=q;
    w:=+st(q); q:+1;
    s:-1; st(s):=q;
    q:=w;
end;

```

```

procedure callsys(entry-1);
begin
    w:=st(q+1);      "old s before program call"
    w:=+st(q); q:+1; "w = entry point addr"
    s:-1; st(s):=q;
    q:=st(w);
end;

```

```

"activation record:
    <heap>
    heaptop: <free space>
    s:      <temporaries>
           <variables>
    b (or g): <line>
    + 1      <old s>
    + 2      <old b>
    + 3      <old g>
    + 4      <old q>
    + 5      <parameters>
           (<function result>)
monitor variable:
    <variables>
    g:      <gate address>
           <parameters>
stacklength = varlength + templength + 5
poplength = paramlength + 4"

```

```

procedure enter(stacklength,poplength,line,
               varlength);

```

```

begin
    x:=s;
    x:=st(heaptop);
    x compare st(q); q:=+1;
    if less<unsigned> then
        goto stacklimit;
    s:=1; st(s):=g;
    s:=1; st(s):=b;
    s:=1;
        st(s):=s;
    st(s):=st(q); q:=+1;
    s:=1; st(s):=st(q); q:=+1;
    b:=s;
    s:=st(q); q:=+1;
end;

```

"error message
will refer to
line of call"

```

procedure exit;

```

```

begin
    s:=b;
    test st(s); s:=+1;
    w:=st(s); s:=+1;
    b:=st(s); s:=+1;
    g:=st(s); s:=+1;
    q:=st(s); s:=+1;
    s:=w;
end;

```

```

procedure enterprog(poplength,line,stacklength,

```

```

varlength);

begin
  increment st(job);
  s:-1; st(s):=g;
  s:-1; st(s):=b;
  s:-1;
    st(s):=s;
  st(s):+st(q); q:+1;
  s:-1; st(s):=st(q); q:+1;
  b:=s;
  g:=b;
  x:=s;
  x:-st(heaptop);
  x compare st(q); q:+1;
  if less<unsigned> then
    goto stacklimit;
  s:-st(q); q:+1;
end;
"error message
will refer to
line 1 of user
program"

procedure exitprog;
begin
  test st(continue);
  if zero
    then goto exception
    else goto terminated;
end;

procedure beginclass(stacklength, 5,line,0);
begin
  x:=s;
  x:-st(heaptop);
  x compare st(q); q:+1;
  if less<unsigned> then
    goto stacklimit;
  s:-1; st(s):=g;
  s:-1; st(s):=b;
  s:-1;
    st(s):=s;
  st(s):+st(q); q:+1;
  s:-1; st(s):=st(q); q:+1;
  b:=s;
  s:-st(q); q:+1;
  w:=st(b+1);
  g:=st(w-1);
end;
"error message
will refer to
line of call"

procedure endclass;
begin
  "same as exit"
end;

procedure enterclass(stacklength, poplength,

```

```

line,varlength);

begin
  "same as beginclass"
end;

procedure exitclass;
begin
  "same as exit"
end;

procedure beginmon(stacklength, 5,line,0);
begin
  x:=s;
  x:=st(heaptop);
  x compare st(q); q:+1;      "error message
  if less<unsigned> then      will refer to
    goto stacklimit;         line of call"
  s:-1; st(s):=q;
  s:-1; st(s):=b;
  s:-1;
    st(s):=s;
  st(s):+st(q); q:+1;
  s:-1; st(s):=st(q); q:+1;
  b:=s;
  s:=st(q); q:+1;
  w:=st(b+1);
  g:=st(w-1);
  st(kernelop):=initgatel;
  st(kernelargl):=g;
  kernelcall;
end;

procedure endmon;
begin
  st(kernelop):=leavegatel;
  st(kernelargl):=st(q);
  kernelcall;
  s:=b;
  test st(s); s:+1;
  w:=st(s); s:+1;
  b:=st(s); s:+1;
  g:=st(s); s:+1;
  q:=st(s); s:+1;
  s:=w;
end;

procedure entermon(stacklength, poplength,
line, varlength);
begin
  x:=s;
  x:=st(heaptop);
  x compare st(q); q:+1;      "error message

```

```

        if less<unsigned> then      will refer to
            goto stacklimit;        line of call"
        s:-1; st(s):=g;
        s:-1; st(s):=b;
        s:-1;
            st(s):=s;
        st(s):=st(q); q:+1;
        s:-1; st(s):=st(q); q:+1;
        b:=s;
        s:-st(q); q:+1;
        w:=st(b+1);
        g:=st(w-1);
        st(kernelop):=entergatel;
        st(kernelarg1):=st(g);
        kernelcall;
    end;

procedure exitmon;
begin
    "same as endmon"
end;

procedure beginproc(line);
begin
    st(b):=st(q); q:+1;
end;

procedure endproc;
begin
    st(kernelop):=endprocess1;
    kernelcall;
end;

procedure enterproc(stacklength, poplength,
                    line, varlength);
begin
    x:=s;
    x:-st(heaptop);
    x compare st(q); q:+1;      "error message
    if less<unsigned> then      will refer to
        goto stacklimit;        line of call"
    s:-1; st(s):=g;
    s:-1; st(s):=b;
    s:-1;
        st(s):=s;
    st(s):=st(q); q:+1;
    s:-1; st(s):=st(q); q:+1;
    b:=s;
    s:-st(q); q:+1;
    g:=st(g+3);
    clear st(job);
end;

```

```

procedure exitproc;
begin
  s:=b;
  test st(s); s:=+1;
  w:=st(s); s:=+1;
  b:=st(s); s:=+1;
  g:=st(s); s:=+1;
  q:=st(s); s:=+1;
  s:=w;
  increment st(job);
end;

procedure pop(length);
begin
  s:=+st(q); q:=+1;
end;

procedure newline(number);
begin
  st(b):=st(q); q:=+1;
end;

procedure incrword;
begin
  increment st(st(s)); s:=+1;
end;

procedure decrword;
begin
  decrement st(st(s)); s:=+1;
end;

procedure initclass(paramlength,distance);
begin
  w:=st(q); q:=+1;           "w=paramlength"
  if nonzero then
    begin
      x:=s;
      x:=+w;                 "x=s+paramlength"
      x:=st(x);
      test st(x); x:=+1;     "x=class addr+1"
      iterate w times
        st(x):=st(s); x:=+1; s:=+1;
    end;
  w:=q;
  w:=+st(q); q:=+1;
  s:=+1; st(s):=q;
  q:=w;
end;

procedure initmon(paramlength,distance);

```

```

begin
    "same as initclass"
end;

procedure initproc(paramlength,varlength,
                    stacklength,distance);
begin
    st(kernelop):=initprocess1;
    st(kernelarg1):=st(q); q:=+1;
    st(kernelarg1):=st(q); q:=+1;
    st(kernelarg3):=st(q); q:=+1;
    st(kernelarg2):=q;
    st(kernelarg2):+st(q); q:=+1;
    kernelcall;
    test st(s); s:=+1;
end;

procedure pushlabel(distance);
begin
    s:-1; st(s):=q;
    st(s):+st(q); q:=+1;
end;

procedure callprog;
begin
    w:=q;
    q:=st(s);
    test st(q); q:=+1;
    st(s):=st(q); q:=+1;
    q:=+2;
    st(s):+q;
    s:-1; st(s):=w;
end;
    "w=old q"
    "q=code addr"
    "st(s)=codeleng"
    "q=codeaddr+4"
    "st(s)=constadr"
    "push(old q)"

procedure truncreal;
begin
    w:=<real> st(s); s:=+2;
    s:-1; st(s):=trunc(w);
    if overflow then
        goto overflowerror;
end;

procedure absword;
begin
    test st(s);
    if negative then
        begin
            st(s):=-st(s);
            if overflow then
                goto overflowerror;
        end;
end;
end;

```

```

procedure absreal;
begin
  st:=abs<real>(st(s));
end;

procedure succword;
begin
  increment st(s);
end;

procedure predword;
begin
  decrement st(s);
end;

procedure convword;
begin
  w:=conv(st(s)); s:=s+1;
  s:=s-2; st(s):=<real>w;
end;

procedure empty;
begin
  clear w;
  test st(s);
  if zero then
    increment w;
  st(s):=w;
end;

procedure attribute;
begin
  w:=st(s);
  st(s):=st(w+head);
end;

procedure realtime;
begin
  st(kernelop):=realtimel;
  kernelcall;
  s:=s-1; st(s):=st(kernelarg1);
end;

procedure delay;
begin
  st(kernelop):=delaygatel;
  st(kernelarg1):=st(g);
  st(kernelarg1):=st(s); s:=s+1;
  kernelcall;
end;

```



```

procedure continue;
begin
  st(kernelop):=contgatel;
  st(kernelarg1):=st(g);
  st(kernelarg1):=st(s); s:=s+1;
  kernelcall;
  s:=b;
  test st(s); s:=s+1;
  w:=st(s); s:=s+1;
  b:=st(s); s:=s+1;
  g:=st(s); s:=s+1;
  q:=st(s); s:=s+1;
  s:=w;
end;

procedure io;
begin
  st(kernelop):=iol;
  st(kernelarg3):=st(s); s:=s+1;
  st(kernelarg1):=st(s); s:=s+1;
  st(kernelarg1):=st(s); s:=s+1;
  kernelcall;
end;

procedure start;
begin
  st(continue):=1;
end;

procedure stop;
begin
  st(kernelop):=stopjobl;
  st(kernelarg1):=st(s); s:=s+1;
  st(kernelarg1):=st(s); s:=s+1;
  kernelcall;
end;

procedure setheap;
begin
  st(heaptop):=st(s); s:=s+1;
end;

procedure wait;
begin
  st(kernelop):=waitl;
  kernelcall;
end;

terminated:
  st(result):=0;
  goto exception;
overflowerror:

```

```

    st(result):=1;
    goto exception;
pointererror:
    st(result):=2;
    goto exception;
rangeerror:
    st(result):=3;
    goto exception;
varianterror:
    st(result):=4;
    goto exception;
heaplimit:
    st(result):=5;
    goto exception;
stacklimit:
    st(result):=6;
    goto exception;
exception:
    st(line):=st(b);
    test st(job);
    if zero then "insystem"
        begin
            st(kernelop):=systemerror;
            kernelcall;
        end
    else "in job"
        begin
            b:=g;
            s:=b;
            test st(s); s:=s+1;
            w:=st(s); s:=s+1;
            b:=st(s); s:=s+1;
            g:=st(s); s:=s+1;
            q:=st(s); s:=s+1;
            s:=w;
            clear st(job);
        end
    end;
end;

```

The Implementation of Concurrent Pascal on the NCR8200

by

Donald Mounday

**B. S., Fort Hays Kansas State College
Hays City, Kansas 1969**

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

**KANSAS STATE UNIVERSITY
Manhattan, Kansas**

1978

ABSTRACT

The programming language Concurrent Pascal has been implemented on various computers since its invention. The subject of this report is the implementation of Concurrent Pascal on the NCR8200. The process of transporting the Interdata 8/32 implementation to the NCR8200 is described along with a discussion of the problems encountered. Improvements to the NCR8200 implementation are also presented. These are the adaptation of the Pascal virtual code to a word addressable architecture and the development of a generalized IO architecture for the Concurrent Pascal kernel.