An investigation of the modula programming language.

Nelson LeRoy Fernandez

Follow this and additional works at: https://preserve.lehigh.edu/etd

Part of the Programming Languages and Compilers Commons

Recommended Citation
https://preserve.lehigh.edu/etd/1968

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.
AN INVESTIGATION OF THE MODULA PROGRAMMING LANGUAGE

by

Nelson LeRoy Fernandez

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computing Science

Division of Computing and Information Science
Department of Mathematics

Lehigh University
1982
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

MAY 14, 1982

Professor in Charge

Head of Division
# TABLE OF CONTENTS

An Investigation of the MODULA Programming Language

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Abstract</td>
<td>1</td>
</tr>
<tr>
<td>II. Introduction</td>
<td>2</td>
</tr>
<tr>
<td>III. Modula</td>
<td>5</td>
</tr>
<tr>
<td>IV. A Modula Program</td>
<td>22</td>
</tr>
<tr>
<td>V. Discussion</td>
<td>34</td>
</tr>
<tr>
<td>VI. Conclusion</td>
<td>41</td>
</tr>
<tr>
<td>VII. References</td>
<td>43</td>
</tr>
<tr>
<td>VIII. Appendix A</td>
<td>44</td>
</tr>
<tr>
<td>IX. Appendix B</td>
<td>55</td>
</tr>
<tr>
<td>X. Vita</td>
<td>61</td>
</tr>
</tbody>
</table>
ABSTRACT

This thesis investigates the system's programming language, Modula. Modula is a computer language recently developed by Nicklaus Wirth, the developer of Pascal. Modula is designed as a high level language used to implement operating systems and real-time programming for device control.

The thesis first explains Modula's syntax and semantics. It then fully describes the theory and operation of some Modula programs. Finally, it describes some implementation details of Modula and discusses how Modula has been received by the academic community.
CHAPTER 1
INTRODUCTION

In the 1940's and early 1950's the electronic digital computer was new to the world. The programmers were scientists who wrote their code in binary without the help of mnemonics. It did not take long for someone to develop the mnemonics for an assembly code with an associated assembler. The abstraction process was furthered with the natural evolution to high level programming languages. An early one was FORTRAN. Some people noted structural problems with Fortran and devised radical approaches in languages, such as ALGOL and LISP. In retrospect, one can appreciate this evolution from complete dependence upon a machine to a well structured abstraction.

At first these language defined processes were one on one, man versus machine. This wasted much machine time. It was, therefore, decided to write an assembly code program to implement an automatic computer operating system so that several users could simultaneously access the processor. There are several criteria for operating systems e.g. small size, efficient code, low overhead on processor time, deterministic output of user programs from one run to the next and reliability in terms of its ability to avoid deadlocks. Initially, these criteria could only be met
through the use of assembly language. At first, exact criteria for operating systems were not well understood. System designers were overly ambitious and their efforts frequently created more problems than were solved. At present, mechanics and abstractions needed to implement operating systems are fairly well known. However, because of an operating system's need for concurrency and speed, operating system construction remains the last bastion of assembly language programmers.

Niklaus Wirth, the creator of the programming language Pascal, in the late 1960's, saw the need for high level construction of operating systems. In an attempt to satisfy this need, he developed the programming language, Modula, in the mid 1970's. The purpose of Modula is to allow operating systems and process control systems, (the latter sometimes called real-time programming) to be programmed at an abstract level. This new language improves system programmer throughput by reducing machine dependence and increasing the capability of organizing system programming into high level abstractions.

Modula is an offspring of Pascal in style and structure. Wirth made minor modifications to Pascal's syntax in constructing the sequential part of Modula. He implemented the current state of the art in designing Modula's approach to concurrency. He also added innovations
of his own in Modula's approach to input and output as well as the elimination of a system clock. While the nucleus of Modula is small, it provides a rich set of operations. Some details such as real numbers, extraneous to systems programming, were omitted.

This paper is an investigation into Modula. It describes Modula's syntax and semantics, gives Modula examples and explanations, and reports on the current work in modifying Modula. Some differences of opinion by other researchers are noted.
One might question why a language such as Modula is needed. Why can one not implement an operating system using Pascal? The answer is the lack of concurrency in Pascal. Pascal has been used to code some operating systems in single user applications on microprocessors. However, when an application involving multiple processors or one processor performing multitasking is required, Pascal cannot be readily used because of its sequential nature.

The following example borrowed from Per Brinch Hansen\textsuperscript{9} assumes an extended version of Pascal which contains a construct allowing concurrency. Despite this additional construct (COBEGIN-COEND), it is still seen that the problem of process synchronization is not easily solved.
VAR Free: Boolean;
Begin
    Free: = TRUE;
    COBEGIN
    process A REPEAT
        REPEAT UNTIL free;
        Free: = FALSE;
        use resource;
        free: = TRUE;
    FOREVER
    process B REPEAT
        REPEAT
            REPEAT UNTIL free;
            free: = FALSE;
            use resource;
            free: = TRUE;
    FOREVER
    COEND
    END.

As noted above, the COBEGIN and COEND in the example indicate concurrency, that is, process A and process B may execute at the same time. For instance, given two processors, one could be devoted to A while the other is devoted to B. One might also have one processor performing multitasking, multiplexed between the two processes. The problem of synchronization is apparent. At the beginning, Free is TRUE. Suppose both process A and process B execute the REPEAT UNTIL free statement simultaneously. They would both set Free to FALSE and attempt to use the resource simultaneously. This would not work if both processes also modify the resource. The processes were intended to have mutually exclusive use of the resource. Most modern day high level languages do not provide easy synchronization techniques with the result that the programming of operating
systems is often done in assembly code.

With the advent of an inexpensive microprocessor, the networking of microprocessors is becoming commonplace. Real-time control applications are limited only by imagination and the time required to do all the necessary programming for operating systems in assembly languages is simply not available. The development of Modula is a natural outgrowth of the necessity to deal with this complexity. We begin with a review of Modula's facilities for sequential programming. Syntax diagrams as given by Wirth are in Appendix A.¹

Modula's sequential programming facilities to a great extent mimic those of Pascal. We, therefore, report them in generality without much detail. Deviations from Pascal will be noted.

The syntax and meaning of constant declarations, is similar to that of Pascal. However, the type real does not exist, and a new type bits is introduced. If a constant is defined to be equal to a string, then that constant is an array l:n of char where n is the length of the string.

The basic types integer, boolean, char, and bits are supported. As noted above, no type real exists. The type bits is an array length L of boolean elements where L is defined as the wordlength of the processor. The values of
an element A of type bits can be set in the program block or as a constant. This is done by enumerating those elements whose value is true and also by using the notation m:n, where all elements between and including m and n are set true. For example, the constant A = [1,2,4:6] sets array elements 1,2,4,5,6 true in A. Types may be enumerated, arrays and record structures (without variants) may be declared, and variables defined (without subranges) as in Pascal.

A deviation from Pascal exists in that the SET type does not exist. This is replaced somewhat by the type bits. Also the Pascal provisions for FILE and pointer do not exist. Wirth felt that Modula would be used to implement file handling and wished to keep the run-time support library small.

The hierarchy for evaluating expressions is as follows:

most precedence
* ,/ ,div ,mod ,and
+ , - , or , xor
least precedence
= , <> , <= , < , > = , >

The operators perform the operations one expects with division being truncated to the integer part and xor standing for exclusive or. It might be noted that, contrary to most implementations of Pascal, when an expression such as (A and B) is evaluated if A is false, B is not evaluated.
The Pascal-like statements supported include the assignment statement, the procedure call with both variable and constant parameters, a slightly modified IF-THEN-ELSE statement, a CASE statement with a required begin-end bracketing the statement following the colon, and a WITH statement. Notice that the GOTO statement, labels and the FOR statement are not included in Modula. An example showing the modified IF statement and the CASE statement follows:

```pascal
IF A=0 THEN
  B:=5
ELSEIF A=3 THEN
  B:=8
ELSE B:=23
END;
CASE A OF
  3 : BEGIN Z:=13 END
  8 : BEGIN Z:=38 END
END;
```

A new statement has been added which is a very general LOOP statement. It allows the programmer to exit the loop on several exit conditions from any place in the loop. An example follows:

```pascal
LOOP
  A:=3; B:=4 WHEN Z=1 DO B:=9 EXIT
  C:=8 WHEN Z=Z EXIT
  Z:=ZDIV 2
END;
```

Both proper procedures and function procedures are supported by Modula. Function procedures are distinguished by "::" type following the parameter list. In a parameter
list, the parameters are either constant or variable as in Pascal. The word CONST specifies a constant parameter, but as in Pascal, in default, the parameter is assumed constant. Recursion is allowed.

Modula varies from Pascal with a nice feature in procedures and modules at nesting level 0. It allows initialization of variables local to a procedure or module. Initialization directly follows the declaration part of a block and immediately precedes the statement part. Procedures also have an optional USE list which, if included, specifies all those objects external to the procedure that the procedure may access. The default is the Pascal convention. One final note on procedures, which some might think unnecessary and others consider good documentation, is that instead of a semi-colon after the last END statement for the procedure as in Pascal, Modula requires the procedure identifier.

This brings us to the MODULE. A module may be thought of as the security fence around its domain. It specifies which objects from the outside that the inside may use through a USE list, and it also specifies which objects the outside may use from the module through a DEFINE list. All structural details of an object in the use list are withheld from the outside. This implies that only procedures exported from the module may use the exported variables or
their structure must be exported explicitly. The variables exported are read-only outside the module. A module may have its own statement block. It is executed each time the procedure to which the module is local is called. The module does not determine the life-span of its variables, only their scope. The procedure to which a module is local determines its variables life-span. Finally, a Modula program has the following syntax:

```
program = module "."
```

Now we introduce the concept of a PROCESS. A process helps to implement concurrency. It must be declared at level 0 and cannot be nested. Its syntax is similar to a procedure except that it uses the word PROCESS in place of the word PROCEDURE. If the process is in a device module talking to a device, its header must include an interrupt vector. When a call to a process is made, execution of the process takes place concurrently with the continuation of the calling statement sequence. When the end of execution of the process is reached, the process goes out of existence. Processes may not create other processes. A process may be initiated more than one time so that multiple versions can be running simultaneously.

What happens if several processes wish to access a common variable? This results in the synchronization problem mentioned earlier where both processes tried to talk
simultaneously with the device. A way of avoiding this is through the implementation of critical sections. The latter only allow one process wishing to access the resource to execute at a time. The other processes wait. Modula's implementation of this is through the INTERFACE MODULE, which is a structure similar to the monitor of Per Brinch Hansen. More than one process may have called a module's procedure and be "within" an interface module at any given point in time, but only one may be executing, the others must be either waiting for a signal or sending a signal or waiting for an interrupt. Procedures declared within an interface module can only call other procedures declared within the same interface module.

Processes use signals to facilitate communication with each other. A signal is declared in the same manner as a variable with the type SIGNAL. It is not a variable in the sense that it can be assigned a value. A signal is used for timing, particularly within interface modules. Typically a process cannot execute before a certain condition is true. If the condition is false, the only way for the condition to become true is for another process to enter its interface module. Then the first process must wait until the other process sets the condition true. This is done with the WAIT(S,R) command. The process waits until the signal S is received. When the signal S is received, the process may
continue. The R parameter is the rank of the call. The higher the rank of the call, the less critical the wait is. Its rank is a positive integer and if omitted, is assumed to be one. All waiting processes of rank n on signal S will be resumed before any of rank n+1 will resume.

Once our first process has entered the critical region, it must inform the rest of the processes when it is finished in case any others are in the wait state. This is accomplished with the SEND(S) command which pulses the signal S and allows the longest waiting process at the lowest rank to resume execution into the critical region. If, prior to so ending signal S, the first process wishes to see if any other process was waiting on S it can use the command AWAITED(S). This command returns the value true if another process is waiting on S and false otherwise.

A signal should not be confused with a semaphore. A signal has no value but is analogous to an electrical pulse whereas a binary semaphore is an electrical pulse followed by a voltage level change (off to on). A binary semaphore may be described as follows:

```plaintext
TYPE semaphore = RECORD
    switch : BOOLEAN;
    free  : SIGNAL
END;
```

In the beginning of this chapter, we presented an algorithm that showed why a language such as Modula is

- 13 -
needed. Now we present the intended function of the previous code in a Modula program.

```
MODULE Sample;
PROCESS A;
BEGIN
  LOOP
    open;
    "use resource";
    release
  END
END A;
PROCESS B;
BEGIN
  LOOP
    open;
    "use resource";
    release
  END
END B;
INTERFACE MODULE reserve;
DEFINE open, release, init;
TYPE semaphore = RECORD
  switch : BOOLEAN;
  free : SIGNAL
END;
VAR S : semaphore;
PROCEDURE open;
BEGIN
  IF s.switch := true
  END open;
PROCEDURE release;
BEGIN
  s.switch := false;
  IF AWAITED (s.free) THEN SEND (s.free) END
END release;
PROCEDURE init;
BEGIN
  s.switch := false
END init;
END reserve;
BEGIN
  init;
  A; B
END sample.
```

This program uses the semaphore S to reserve the
resource. The procedure `open` checks if the resource is free; if not, it waits for a signal, if it is, it reserves the resource. The procedure `release` changes the boolean element to false and signals `s_free` if another process is waiting. The procedure `init` just initializes `s_switch`. The program first calls `init` and then it calls process's A and B which execute simultaneously. Notice that the main program and processes A and B are allowed to use `open`, `release` and `init` because they are in the define list of the interface module.

There are a couple of modifications that can be made which will not change the logic. First, the check for `awaited` is not necessary since sending a signal does no harm if nothing awaits it. The line could, therefore, be as follows:

\[
\text{SEND (s_free)}
\]

Second, when the procedure `release` does perform a send according to the scheduler, the waiting process is continued, and the process executing procedure `release` is suspended until such time as the other process leaves the interface module. In this example, the first thing a process does in procedure `open` after its wait is to set `s_switch` to true. Therefore, `s_switch` did not have to be set to false in procedure `release` with this execution
sequence. Now if one takes into consideration that only one process may execute in the interface module at a time, then the two lines of release:

\[
s'\text{switch}:=\text{false}; \\
\text{IF AWAITED (s'free) THEN SEND (s'free) END}
\]

can be changed to:

\[
\text{IF AWAITED (s'free) THEN SEND (s'free) ELSE s'\text{switch}:=\text{false END}}
\]

Of course, with all programming languages, especially ones used to implement operating systems, a time must come when it has to leave the totally abstract world and become involved with hardware. This is done in Modula with a so-called DEVICE MODULE and its associated device processes or drivers. A device module is declared as such, and the device processes can only be declared within the device module. Device processes are not syntactically distinguished from regular processes which cannot reside within a device module. The processes within a device module drive peripheral devices. The only place the command DOIO may be used is within a device process. The DOIO statement, usually following an enable interrupt and device initiator, is the part of the code which represents the execution of the peripheral device. Thus the device process is maintained until the peripheral device finishes and sends back an interrupt. Hence, when a process performs a doio,
it is held in a state much like a wait state allowing other processes that have called local procedures to access local variables.

In the heading of a device module there is a number in brackets. This is called its interrupt priority level. Regular processes have level 0. Device processes have a higher level. Interrupts to the processor are recognized when it executes a doio, send, or wait command. The processor first services the highest priority interrupt that has waited longest. Interrupts from processes at a lower level are disabled and saved until all process interrupts at the higher level are serviced. After this, the processor drops to the next highest saved interrupt. Should a device process send a signal to a process of lower priority (a regular process), the device process continues. This deviates from the discussion of signals which stated that the signaled process resumes execution. This feature of Modula is hardware dependent in that the interrupt facility differs from machine to machine, and some processors have no interrupts.

Earlier we mentioned the interrupt vector in the heading of a device process. This is the actual interrupt location of the device driven by the particular process. As is common, this necessitates one device process or driver per device. Those interrupts are disabled during the
execution of a wait statement.

Wirth lists several restrictions on device processes. They are:

1. They must not send signals to other devices processes.

2. They must not call any non-local procedures.

3. Only a single instance of a device process can be activated. They are not re-entrant.

4. Wait statements within a device process cannot specify a rank.

For a process to communicate with a device, certain areas of memory must be allocated for tasks such as buffering of I/O or checking of device status. These areas are called device registers. Their implementation is necessarily hardware dependent. Declaration of a device register is the same as a variable except with the added address in brackets. An example follows:

```
REG [1800B] : char;
```

If the device register is used to get a device status, it should be declared of type \texttt{bits}. If the register is a device register for buffering of I/O, it should probably be declared of type \texttt{char}. This scheme for device control and
I/O is readily implemented on several processors including the DEC PDP-11, INTEL 8080, and the TEXAS TMS 9900.

This concludes the description and explanation of the language Modula. The following is a summary of its variances from Pascal.

1. No subrange type declarations.
2. No variant record declarations.
3. No file type and its corresponding heap for memory allocation.
4. No type pointer.
5. No set type but a bits type. No labels or goto branching.
6. No FOR loop, but a general loop structure.
7. No type real or real arithmetic.
8. Addition of the following constructs: module, process, signal, interface module, and the device module with its related device process or driver.
9. A define list and a use list for documentation and protection, with read-only exported variables from modules.
10. Slight modification syntactically of some statements, such as the if-then-else and the case statement.

11. Modula includes the following standard procedures.

   a. inc \((x,n)\) performs \(x := x + n\)
   
   b. dec \((x,n)\) performs \(x := x - n\)
   
   c. inc \((x)\) performs \(x := x + 1\)
   
   d. dec \((x)\) performs \(x := x - 1\)
   
   e. halt performs termination of the program
   
   f. off(bl,b2) sets bl and b2 = [] with bl,b2 = type bits
   
   g. off(bl) sets bl = [] with bl = type bits
   
   h. among \((i,b)\) equals \(b[i]\) with b = type bits
   
   i. low \((a)\) equals the low index bound of array a.
   
   j. high \((a)\) equals the high index bound of
array a.

k. adr (V) equals the address of variable V.

l. size (V) equals the size of variable V.

m. integer (x) equals the ordinal of x in the set of values defined by the type of x. For example, if x is type char in the ASCII character set then integer ('A') = 65.

n. char (x) equals the character with the ordinal of x. This function is also supported in Pascal. For example char (65) = 'A'.
CHAPTER 3
A MODULA PROGRAM

With the syntax and semantics of Modula described, this chapter will put the concepts together in a program, given by Wirth, and explain exactly how it works. The program is written to demonstrate the nice features and concepts of Modula with the proviso that it be understood that it is not an entire, real system but a model.

The model system involves the following hardware: a card reader, a line printer, a typewriter printer, a keyboard, and a processor with a clock. The system has two data streams. The first takes input from the keyboard and sends it to the typewriter printer. This might be considered the implementation of a full duplex terminal. The second data stream reads input from the card reader and sends it to the line printer.

The data streams are completely independent of one another. Each data stream could be modeled as follows:

1. Data comes in through a device driver which places it into an input buffer.

2. The main program transfers data from the input buffer to the output buffer.
3. The output device driver sends data in the output buffer to the output device.

Though totally uncoupled, the two data streams will have many similarities. The buffer is the communications link between the device and the processor. Therefore, for each data stream we need two device modules and their associated device drivers. This is to be contrasted to the use of, one interface module with the get and put procedures within it for device driver independent buffer transfer.

With the device-to-computer transfer of data, the driver will store data into the buffer, and the computer will remove it. This means there must be a procedure local to the device module that the computer can call and which will remove one character from the buffer. The reason it should reside within the device module is to provide mutual exclusion of the two actions from the buffer. The same criterion also applies to the output buffer. Thus, we have four device modules each with a driver and a procedure.

The program is given in Appendix B. It is more complicated than described, but its major structure is as described. The rest of the program is concerned with hardware dependent implementation details. The independence of the data streams allows one to investigate the details of each one separately.
The first data stream, the keyboard to the typewriter is the easier case. Both devices use ASCII, they work on a single character basis, and don't require a status interrogation. Therefore, the main process needs only to get a character and then put it. If the character is a carriage return, then it puts the carriage return as well as a line feed. This defines process stream.

The two device modules are called keyboard and typewriter. The module keyboard has the procedure get (var: ch: char) and its device process called driver. It has a circular character buffer size n (n=16) called buf, a variable nf indicating the number filled in buf, two variables inx and outx which specify the location of the next character to be put into buf and removed from buf, and two signals nonfull and nonempty. A send (nonfull) is executed each time the procedure get removes a character frombuf. A send (nonempty) is executed each time the driver puts a character into buf.

An execution process is as follows:

1. The main program is activated. This also activates any statement blocks of modules local to it. Hence, within the keyboard module, inx and outx are set equal to one, nf is set equal to zero, and the process driver is executed.
2. the process stream1 is executed by the main.

3. Stream1 calls get(ch); procedure get is local to the keyboard module.

4. If driver is executing a send, wait, or doio, then procedure get is executed. If nf = 0, then there has been no input so procedure get must wait for input, otherwise, procedure get sets ch equal to the character in buf pointed out by outx. It then increments outx by one modulo the size of buf, it subtracts one from nf, and signals that buf cannot be full.

5. The driver process runs continuously. It has two variables for communicating with the keyboard. The first, kbs, is the keyboard status and the second, kbb, is the keyboard buffer (a one character buffer). If nf equals n, then buf is full, so it waits for a signal that it is not full. The command kbs[6]:=true enables interrupts from the keyboard. These are followed by a doio waiting for the character. Upon receipt of the character, an interrupt is generated. This brings the processor to the command kbs[6]:=false. This disables interrupts. One does not want more characters to be interrupting before the character received has been
processed. The input \texttt{kbb} is divided modulo 200B to assure an ASCII character representation before the char of the result is put into \texttt{ch}. The check for \texttt{ch} equal to the escape character provides a way to end the program. The input \texttt{ch} is then put into \texttt{buf} and its pointer is incremented. The total number filled in the buffer is incremented and a signal is sent saying the buffer is not empty.

6. The process \texttt{streaml} upon receipt of a character continues.

Notice that the interrupt address for \texttt{driver} is 60B. The address of \texttt{kbs} is at 177560B and that of \texttt{kbb} is at 177562B. The interrupt priority of the device module is 4.

The \texttt{typewrite} device module is very similar in behavior to the \texttt{keyboard} device module. Initially the buffer is set empty and the process \texttt{driver} is started. The only real difference between the two modules is that \texttt{driver} is emptying the buffer and sending characters to the typewriter while the procedure \texttt{put} is filling the buffer. Most of the variables in the \texttt{typewrite} module are named in the same manner as in the \texttt{keyboard} module. They are not physically the same variables, since they are declared locally. Notice that for the device modules, the "define" statement requires that the only inner objects that the rest of the program can
access is procedure put for the typewrite module and
procedure get for the keyboard module.

The data flow of stream2 is somewhat more complicated
than that of stream1 due both to hardware peculiarities and
software sophistications. The hardware problems are
fourfold. First, the card reader delivers data encoded in
one scheme, while the line printer accepts ASCII encoding.
This implies that a translation must be performed. Second,
the card reader when activated sends a full card worth of
data, namely eighty characters plus an end-of-card flag.
Third, the high rate of data transmission makes it wasteful
to send synchronization signals after each character. Thus,
data is blocked in groups with synchronization signals being
sent after transmission of a whole block. Last, the card
reader's status must be interrogated both before reading a
card to determine if it is ready and after reading each
character to determine if it is the end of the card. The
card reader does not provide an interrupt when it goes from
a not ready to a ready state, which means its status must be
polled periodically. The latter necessitates a clock.

The fact that the card reader sends data in spurts of
eighty characters plus an end-of-card marker means that
before a card can be read, at least eighty-one spaces must
exist in the buffer. This buffer is, as before, a circular
buffer called buf, but its size is 256. The following
variables and signals have the same meaning as in the keyboard module: \texttt{inx}, \texttt{outx}, \texttt{nf}, \texttt{nonfull}, and \texttt{nonempty}. The only addition is the variable \texttt{ne}, which stands for the number of empty slots available in the buffer. This means that \texttt{ne} must be greater than or equal to eighty-one before a card is read.

The procedure \texttt{read} (var \texttt{x:integer}) is similar to the procedure \texttt{get} with one exception. The variable \texttt{ne} is incremented each time a character is sent from the buffer and a signal is sent only if this makes \texttt{ne} greater than or equal to zero. This is necessary because the process \texttt{driver} reserves room for eighty-one characters. If this makes \texttt{ne} less than zero, it waits for the signal that tells it that enough space is available, that is, \texttt{ne} is greater than or equal to zero.

The process \texttt{driver} is also very similar. It decrements \texttt{ne} by eighty-one characters to reserve room. If the room exists (\texttt{ne} \geq 0), then it continues, otherwise, it waits for the signal from procedure \texttt{read}. The check of \texttt{crs} (card reader status) bits 8 and 9 is required to see if the card reader is ready. When the reader is made ready by manually pushing a reset button, no interrupt is generated. This means that the computer must periodically poll the device until it is ready. This is the \texttt{wait} (crsig), where \texttt{crsig} is a timing signal generated indirectly by a clock.
command \texttt{crs:=[0,6]} both sets an interrupt enable and starts the card motion. It next performs a doio. The statement checking \texttt{crs[14,15]} is true when the end of the card is reached. If it isn't true, the procedure \texttt{put} is called to place the device buffer character into array \texttt{buf}. Note that this is all done without any \texttt{waits} or \texttt{sends}. Thus, the attempt is made to let the process continue uninterrupted as it reads a card. Within this loop the only time another process may access module variables by calling procedure \texttt{read} is during the execution of the DOIO statement. When a complete card is read, a -1 is put in \texttt{buf} as an end-of-card flag, and the device interrupt is disabled. A signal is sent if \texttt{buf} has characters to be read.

In order to generate the clock signal \texttt{crsig}, a hardware clock must be accessed. To do this, there exists a device module called \texttt{timing}. It does nothing more than enable an interrupt to the hardware clock, perform a doio, and send a signal called \texttt{tick}. It is done in a loop so that in Wirth's system a signal was generated every 20 nsec.\textsuperscript{2} This signal is made available through the define statement. The card reader driver cannot use this signal due to the restriction stated earlier. This restriction states that no device process may send signals to another device process. This means that the regular process called \texttt{clock} does no more than await the signal \texttt{tick} and then generates the signal
crsig used by the card reader driver.

The process stream2 first reads an integer x, which is the input from the card reader. If this x is equal to eoi (end of input), then it flushes the buffer until it gets a -1, the end-of-card flag, and causes a form feed. The eoi is a special 7-8-9 punch in column 1 of the last card. Otherwise, the process performs a loop in which it checks for the end of a card. If the end-of-card has not occurred, it converts the number x to an ASCII character ch, writes it to the line printer buffer, and reads the next integer x. When the loop is terminated, it is at the end of a card and a line feed is written to the buffer.

In order to perform the translation to ASCII, an array is filled with the ASCII characters. An algorithm in procedure convert is used to perform the conversion of x to ch.

The line printer process is similar to that of the card reader except that its data is received in varying sized blocks. It also has a hardware buffer. This means that it accepts up to one line's worth of data before it prints. This allows the driver to place data into the buffer without waiting for interrupts. This continues until a line feed is sent or the buffer is full. Then the doio statement is executed while the line printer prints. Looking at the
driver within device module line printer first, the line printer buffer is set equal to the ASCII control character dc3. This activates the chain in a chain printer. This will be discussed in more detail later. If nf<0, the buffer is empty, and it waits for the signal nonempty. It then accepts characters one at a time and deposits them in the line printer buffer. The only check made is to determine if buffer bit 7 is not set. If bit 7 is not set, the printing is taking place and the doio is executed with the interrupt enabled so that the line printer can interrupt when it is finished. If ch=1f, then the printing has taken place. The driver sends a signal if it has incremented ne to a greater than or equal to zero state.

The procedure write is a familiar procedure. There is a procedure weol, which is executed by process stream2 at the end of each line. It increments nf by one and sends the signal nonempty to the driver. The variable nf is a line counter for stream2. The procedure write is used to fill the buffer with a whole line before alerting the driver to process it by calling weol (write when at end of line).

The software sophistication mentioned earlier involves the driving of a chain printer. The chain wears over time. Therefore, it was decided that the chain action should be turned off if the printer is not used for a period of more than ten seconds. This is done by cooperation between the
driver, a procedure `testchain`, and a process `chaincontrol`.

The process `chaincontrol` spins in a loop calling procedure `testchain` and then waiting for a clock signal which comes every 20 nsec. The procedure `testchain` checks if \( nf > 0 \). If so, the printer is either active or soon will be. The procedure `testchain` waits on the signal `guard` which the driver sends when it becomes inactive. If \( nf < 0 \), then the counter `del` is decremented by one. It has an initial value of 500 and is decremented at each clock signal. If it reaches zero, then the line printer buffer is set to `dc4`, and the chain drive is stopped. The process then waits for the next time it is needed to stop the chain.

In the driver there is a check to determine if \( nf < 0 \). If the latter holds then there is nothing to print. If so, it sends the signal `guard` to alert the process `chaincontrol` to begin counting, and it waits for something to be put in the buffer. When it gets the nonempty signal, it first must start the chain with `dc3` and then reset the counter for `chaincontrol`. The latter will have been put in a wait state by either the chain having been stopped or by the fact that procedure `weol` incremented `nf`.

To summarize, there are two data streams, one from keyboard to typewriter and another from card reader to line printer. They both communicate via nonempty and nonfull
signals with each data stream having two device modules. The second data stream must also include a clock to periodically check the status of the card reader and also to conserve the running of the line printer chain drive. The main module must start both data streams, the clock, and the chain control, with all processes running concurrently.
CHAPTER 4
DISCUSSION

Now that Modula has been described, and a sample program has been given showing how Modula is intended to be used, one might wish to know how well has Modula been received by others and some of its implementation details.

Some details of implementation are covered in notes by Wirth. In them he notes that the module concept is a superset of the class concept of Simula. It does not have as nice a notation, but it has an easier implementation. The main disadvantage of modules is that they require a multipass compiler. The module borrows from the class concept the notion of hiding the structure of exported variables. The module allows read-only exported variables to prevent programmers from writing procedures that return nothing but the variables' value if no variable could be exported.

In Modula, there is always a fixed number of currently active processes. There is an allowance for allocation of workspace for a new process, but when a process terminates, its store is not recycled. With the restriction of the ability to generate new processes in the main module and with a fixed recursion depth at compile time, the stack needed for each process can be allocated at compile time.
Processes are linked in a ring. This provides for no storage overflow and for no problems in which the son outlives the father in process calls.

There is a piece of code that is identical for all Modula programs. It is called the Nucleus and resides within the computer's main store. For systems programming, the nucleus should be small. A large nucleus introduces overhead and constrains a language's flexibility. Modula has a routine for beginning a process, for the wait statement, for the send statement, and for system initialization in its nucleus. Wirth's implementation of Modula on the DEC PDP-11 kept the nucleus to 98 words. This is small enough to consider supporting in hardware.

The language Modula and its implementation has produced effort to try it on other machines. These trials have brought forth some valid criticism. Reghbati and Hamacher discuss the problem of scheduling. In Modula, the only way a process gives up the processor is by issuing a doio, wait, or send. Also, there is no way to multiplex processes through the processor. Processor scheduling is done with a FIFO stack. The only way a process can be scheduled is by means of a call from the main program.

Reghbati and Hamacher see problems with this scheme and propose added versatility for process control applications.
First, they wish to allow process initiation by both a command from the operator console and by an interrupt from the device being controlled by the program. They also wish to allow for a different scheduling algorithm than FIFO scheduling. They wish to consider a deadline scheduler which involves executing the process which is closest to its deadline (set at process initiation). Of course, all this would increase the nucleus size.

Allchin discusses the problems involved with not having a "system" clock. When a call to a device is made, the device process executes a doio which waits for an interrupt from the device to signify it has completed the task. What if the device does not generate interrupts, or what if something unusual happens and the interrupt is lost. With straight Modula, this would cripple the process. What is needed is the notion of a device timeout, perhaps by adding a parameter to the doio. This would necessitate a system clock to time the device. Another scheme might be employed using two device processes in the device module, where one of them acts as the timer. This is similar to the case of the chain timer for the line printer. Allchin feels that this is a very complicated substitute for a common function in process control systems.

An example of a language with a clock within the nucleus is Concurrent Pascal. Its advantages include the
fact that it guarantees that all processes are given a "time slice" and would allow the scheduler to be part of the nucleus. The scheduler is generally the most time critical part of the system and, therefore, would be more efficient in machine code. Its disadvantages include the aspect that now one can't tailor a clock to meet a particular need. The existence of a clock might cause unnecessary overhead if a clock is not needed. The universal clock would also restrict the scheduling algorithm by not allowing as much fine tuning.

The scheduling algorithm used in Modula essentially relies on individual processes to be gracious enough to occasionally perform a doio, wait, or send to allow processes a time slice. In some cases, another process might conceivably need to pre-empt the active process. In Modula, this is not possible.

Modula also makes certain requirements upon its hardware. Since all processes, except device processes, are re-entrant, the hardware must support a stack. The hardware must also have an interrupt system. The I/O devices must be addressable in one uniform fashion since the compiler will only compile code in one way. Two possible options might be to have either all indirect addressing or all direct addressing to special ports.
Holden and Wand have various complaints and observations about Modula that they formulated while writing a compiler for it. Their nucleus ran to 150 words for standard Modula and to 200 words when they added other operations. This is compared to the 90 words for Wirth and roughly 4,000 words for a Concurrent Pascal nucleus. They felt that the module should be made more like a class so that identical devices would not require separate drivers.

Holden and Wand developed some nice syntax extensions. They felt that provisions should be made for comments as well as to allow an earlier declared constant to be used in the declaration of another. For example:

```
CONST two=2;
   four=two+2;
```

They suggested an ELSE part in the CASE statement to capture the non-enumerated cases. Further, they proposed the enumeration of the values of an array in a single statement. For example, if A is an array of 4 integers:

```
A: = (-1,3,8,4);
```

This concept can be extended to record types. If R is a record with three integers declared within it, and A is the type R, then

```
A: = (1,2,3);
```
sets A'first_integer to equal 1 and so forth.

Bernstein and Ensor found enough fault with Modula to change its structure to another language called SB-MOD. In SB-MOD, variables need not be strictly read-only between modules, but they can also be variable parameters. Further, in SB-MOD facilities are included for building a hierarchial system and for program verification. In addition, the signal is replaced by a "condition" with four operations defined on it: send, wait, awaited, and mark. Condition is declared as follows:

C:CONDITION B [address]

The condition C has optional boolean B and address. The address implies that C is used to interface with a device. This does away with the statement doio, using instead a wait.

The statement \texttt{wait(c,k)} in SB-MOD is the same as that of Modula unless a boolean is declared. Then upon execution, the boolean is evaluated and the wait perform if it is false. The statement \texttt{send(c)} also behaves in a similar manner except when the boolean is false. Then the send does nothing. The function \texttt{awaited} is identical to that of Modula. The operation \texttt{mark(c)} is performed by a process. If the queue of c is empty it does nothing. If it is non-empty, then the process at the head of the queue will
be marked if it has a boolean expression. If there is no boolean expression, it will be put in the ready state. Later when the marking process has relinquished control of the interface module, the booleans of the marked processes are evaluated. If true, then the process continues.

In general, other researchers seem to applaud Modula or a similar concept as a long-needed tool in computer science. They, nevertheless, find many faults with Modula as one might expect of an early attempt at an abstract systems programming language.
CHAPTER 5

CONCLUSION

Modula is a definite step in the right direction. It has not, as yet, evolved into its final state. Due to its probing into a newer area of computer science, it was bound to have faults. Not all of these faults were unrecognized by Wirth but were tradeoffs where two philosophies collided. An example of this is the lack of a system clock.

The three most significant limitations of Modula are the following. First, it is hard to implement on computers that are without index registers, stacks or interrupts. Second, one driver can't service two identical devices, but two identical drivers must be written. Third, processes depend upon other processes to be gracious enough to execute a doio, send, or wait command which allows for Modula's time slicing.

Modula was written for medium to small computer implementation. This is where most Modula work has been done. It has been implemented on such processors as the DEC PDP-11, INTEL 8080, TEXAS TMS9900, and recently a version was released for the Apple II.

Modula's most important concept is the notion of a device module. This confines system details to one very
small portion of the program leaving the programmer free to
deal with abstractions. Even if Modula does not become
widely accepted, its notion of a device module most
certainly will.
REFERENCES


STATEMENTSEQUENCE

STATEMENT

FACTOR

UNSIGNED CONSTANT

VARIABLE

IDENT

( EXPRESSION VARIABLE )

, EXPRESSION

( EXPRESSION )

HOT
TERM

FACTOR

HIV

AND SIMPLEXPRESSION

ZD^TERHJORJ—> XOR I—
INITIAL VALUE

CONSTANT

({ INITIAL VALUE }

INTEGER

IDENT

{ INITIAL VALUE }

{ }

{ }

{ }
module datastreams;
const 1F = 12C: ff = 14C: cr = 15C;
var crsig: signal;
device module timing [6];
define tick;
var tick: signal;
lcs [177546B]: bits: (*clock status*)
process driver [100B];
begin lcs[6]: = true;
loop doio: send(tick)
end
driver;
begin driver
end timing;
device module keyboard [4];
define get;
const n = 16; esc = 33C;
var inx, outx, nf: integer;
nonfull, nonempty: signal;
buf: array 1: n of char;
procedure get(var ch: char);
beg
if nf = 0 then wait(nonempty) end;
ch: = buf[outx]: outx: = (outx mod n)+1;
dec(nf); send(nonfull)
end get;

process driver [60B];
var kbs [177560B]: bits; (*status*)
    kbb [177562B]: integer; (*buffer*)
ch: char;
begin
loop
if nf = n then wait(nonfull) end;
kbs[6]: = true; doio; kbs[6]: = false;
ch: = char(kbb mod 200B);
if ch = esc then halt end;
buf[inx]: = ch; inx: = (inx mod n)+1;
inc(nf); send(nonempty)
end
end driver;
begin inx: = 1; outx: =1; nf: = 0; driver
end keyboard;
device module typewrite [4];
define put;
const n = 64; (*buffer size*)
var inx, outx, nf: integer;
nonfull, nonempty: signal;
buf: array 1: n of char;
procedure put(ch: char);
begin
  if nf = n then wait(nonfull) end;
  buf[inx]: = ch; inx: = (inx mod n)+1;
  inc(nf); send(nonempty)
end put;

process driver [64B];
  var tws [177564B]: bits; (*status*)
    twb [177566B]: char; (*buffer*)
begin
  loop
    if nf = 0 then wait(nonempty) end;
    twb: = buf[outx]; outx: = (outx mod n)+1;
    tws[6]: = true; doio; tws[6]: = false;
    dec(nf); send(nonfull)
  end
end driver;

begin inx: = 1; outx: = 1; nf: = 0; driver
end typewriter;

device module cardreader [6];
define read;
use crsig;
const n = 256: (*buffer size*)
var inx, outx, ne, nf: integer;
  nonfull, nonempty: signal;
  buf: array1: n of integer;
procedure read(var x: integer);
begin
  dec(nf);
  if nf<0 then wait(nonempty) end;
  x: = buf[outx]; outx: = (outx mod n)+1;
  inc(ne); if ne>=0 then send(nonfull) end;
end read;

process driver [230B];
  const m = 81; (*block size*)
var crs [177160B]: bits; (*status*)
  crb [177164B]: integer; (*buffer*)
procedure put(x: integer);
begin
  buf[inx]: = x; inx: = (inx mod n)+1;
  inc(nf)
end put;

begin
  loop dec(ne, m);
    if ne<0 then wait(nonfull) end;
    while not off(crs, [8,9]) do wait(crsig) end;
  crs: = [0,6]; (*start card motion*)
  loop doio;
    when not off(crs, [14,15]) exit
  put(crb)
end;
put(-1); crs[6]: = false; (*end of line mark*)
if nf>=0 then send(nonempty) end
end
driver;
begin inx: = 1; outx: = 1; nf: = 0; ne: = n; driver
end cardreader;

device module lineprinter [4];
define write, weol, testchain;
use if;
const n = 512; (*buffer size*)
dc3 = 23C; dc4 = .24C;
chaindelay = 500; (*10 sec*)
var inx, outx, ne, nf, del: integer;
nonfull, nonempty, guard: signal;
buf: array 1: n of char;
lps [177514B]: bits; (*status*)
lpb [177516B]: char; (*buffer*)

procedure write(ch: char);
begin dec(ne);
if ne<0 then wait(nonfull) end;
buf[inx]: = ch; inx: = (inx mod n)+1;
end write;

procedure weol; (*write end of line*)
begin inc(nf); send(nonempty)
end weol;

procedure testchain;
begin
if nf>=0 then wait(guard)
else dec(del);
fi del = 0 then
  lpb: = dc4; wait(guard)
end
end testchain;

process driver [200B];
var ch: char;
begin lpb: = dc3;
loop dec(nf);
if nf<0 then
  send(guard); wait(nonempty);
  lpb: = dc3; del: = chaindelay
end;
repeat ch: = buf[outx]; outx: = (outx mod n)+1;
  inc(ne); lpb: = ch;
if not lps[7] then
Ipsf61: = true; doio; lpsT^l:
end
until ch = 1f;
if ne>=0 then send(nonfull) end
end
driver;

begin inx: = 1; outx: = 1; ne: = n; nf: = 0; driver
end lineprinter;

process streaml; (*keyboard to typewriter*)
use get, put;
var ch: char;
begin
loop get(ch);
if ch = cr then put(cr); put(cr); put(1f)
else put(ch)
end
end
end streaml;

process stream2; (*cardreader to lineprinter*)
use read, write, weol;
const eoi = 37B; badchar = '?
var x: integer; ch: char;
var t: array 0: 63 of char; (*translation table*)
z: array 0: 7 of integer;

procedure convert; (*x to ch*)
var zone, digits: integer;
begin zone: = x div 32; digits: = x mod 32;
if zone<0 then ch: = badchar else
if digits>=16 then digits: = 9 end;
ch: = t[16*zone + digits]
end
end convert;

begin
z[  0]:=0;  z[  1]:=3;  z[  2]:=2;  z[ 31]:=-1;
z[  4]:=1;  z[  5]:=-1;  z[  6]:=-1;  z[  7]:=-1;
t[  0]:='';  t[  1]:'1';  t[  2]:'2';  t[  3]:'3';
t[  4]:'4';  t[  5]:'5';  t[  6]:'6';  t[  7]:'7';
t[  8]:'8';  t[  9]:'9';  t[ 10]:'0';  t[ 11]:'1';
t[ 12]:'#';  t[ 13]:'@';  t[ 14]:'"';  t[ 15]:'!';
t[ 16]:'&';  t[ 17]:'A';  t[ 18]:'B';  t[ 19]:'C';
t[ 20]:'D';  t[ 21]:'E';  t[ 22]:'F';  t[ 23]:'G';
t[ 24]:'H';  t[ 25]:'I';  t[ 26]:'<';  t[ 27]:'.';
t[ 28]:'J';  t[ 29]:'K';  t[ 30]:'\';  t[ 31]:'';
t[ 32]:'L';  t[ 33]:'M';  t[ 34]:'N';  t[ 35]:'O';
t[ 36]:'P';
end

- 59 -
loop read(x);
  if x = eol then
    repeat read(x) until x<0;
    write(ff); (*form feed*)
  else
    while x>=0 do
      convert; write(ch); read(x)
      end;
    write(lf); weol (*line feed*)
  end
end stream2;

process clock;
  use tick, crsig;
begin
  loop wait(tick); send(crsig)
  end
end clock;

process chaincontrol;
begin
  loop testchain; wait(tick)
  end
end chaincontrol;

begin (*datastreams*)
  streaml; stream2; clock; chaincontrol
end datastreams.
Nelson LeRoy Fernandez was born in Nashville, Tennessee, on January 28, 1958. He is the son of William Nelson Fernandez and Rita Woods Fernandez.

From 1972 through 1976, Mr. Fernandez attended Emmaus High School in Emmaus, Pennsylvania. From 1976 through 1980, he attended Shippensburg State College in Shippensburg, Pennsylvania, graduating cum laude. He was president of the Math Club in his junior year. He was elected to Kappa Delta Pi and Kappa Mu Epsilon honorary societies, serving as president of Kappa Mu Epsilon in his senior year.

From 1980 to June of 1982, Mr. Fernandez was a graduate student at Lehigh University in the Division of Computing and Information Science. In 1980, Mr. Fernandez accepted a position as Member of Staff at Bell Laboratories, Allentown, Pennsylvania. He is married to Toni Jo Fernandez.