Eidgenössische
Technische
Hochschule
Zürich

Institut
für
Informatik

Niklaus Wirth

The use of MODULA

and

Design and Implementation of MODULA

THE USE OF MODULA

N.Wirth

Abstract

Three sample programs are developed and explained with the purpose to demonstrate the use of the programming language Modula. The examples concentrate on the uses of modules, concurrent processes, and synchronizing signals. In particular, they all focus on the problems of operating peripheral devices. The concurrency of their driver processes has to occur in real time. The devices include a typewriter, a card reader, a line printer, a disk, a terminal with tape cassettes, and a graphical display unit. The three programs are listed in full.

Author's address:
Institut für Informatik, ETH, CH-8092 Zürich

THE USE OF MODULA

1. INTRODUCTION

The purpose of this paper is to describe three sample programs written in the programming language Modula, thereby to demonstrate techniques of multiprogramming, and to display the language's flexibility. These examples are by no means parts of "real" systems; on the contrary, the posed problems are constructed to condense typical multiprocessing problems into a nutshell, and in particular to demonstrate the use of Modula's facilities to operate a computer's various devices. The first program uses four devices concurrently: a typewriter keyboard and printer, a card reader, and a line printer. The second program transfers files from a terminal with cassette tapes to a disk store and vice-versa, and the third program operates a graphical display unit.

2. THE PROGRAM "DATASTREAMS"

Our objective is to design a system which represents two data streams. Stream 1 originates at the keyboard and flows to the typewriter printer. Stream 2 leads from the card reader to the line printer. Both streams are sequences of characters (ASCII code), and all four devices are to be buffered. The streams are to flow independently; hence, the system consists of two entirely independent, uncoupled parts. Each part is represented by a system consisting of three processes and has the structure shown in Fig.1.

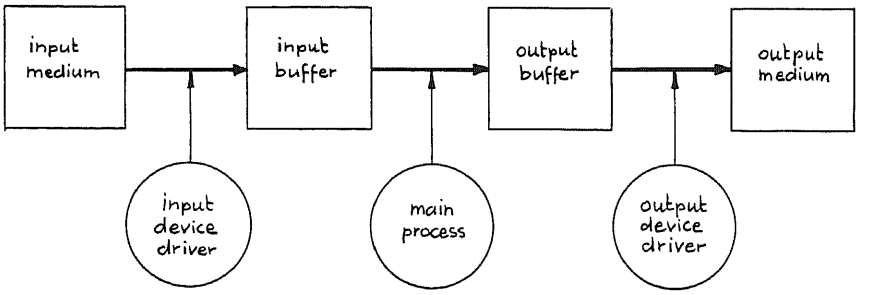

Fig.1, Data flow and system structure

The buffers (and their associated variables) constitute the interfaces between the processes. Hence, it is appropriate to postulate two interface modules. The buffers and the routines depositing and fetching data in and from the buffers are the natural components of each interface module which provides mutual exclusion of interacting processes. Since each buffer is associated with a unique device, and therefore also with a device driver process, the interface module is declared as a device module (whose servicing priority is defined by the PDP-11 hardware). According to the rules of Modula, a process is

declared entirely within the interface module, if it is a driver process.

Normal buffer interface module:

```
interface module M;
   var buffer: T;
   procedure deposit( ); ... (*used by producer process*)
   procedure fetch( ); ... (*used by consumer process*)
end M
```

Buffer interface module, if the consumer process is a driver process:

```
device module M [priority];
   procedure deposit( ); ... (*used by the producer*)
   process consumer;
   begin initialise;
        loop fetch;
             initiate consumer device; doio
        end
   end consumer
end M
```

Buffer interface module, if the producer is a driver process:

```
device module M [priority];
   procedure fetch( ); ... (*used by consumer*)
   process producer;
   begin initialise;
        loop initiate producer device;
             doio; deposit
        end
   end producer
end M
```

2.1. The keyboard to typewriter stream.

This example shows the simplest possible case. Its simplicity is threefold:
1. Both devices use the same encoding of information (ASCII).
2. Both devices operate on the basis of single character transmission.
3. The devices require no status interrogation.

The main process is chosen to be as simple as possible. Its sole obligation is to transfer individual characters from the input buffer to the output buffer. The only additional duty is to recognise carriage return control characters (cr), and to generate an additional line feed character (lf) following each cr.

Each of the two interface modules contains a data buffer; these buffers are used as cyclic buffers. With each buffer are associated a variable n indicating the number of currently filled buffer elements, and two indices (inx and outx) designating the locations of the next characters to be fetched

and deposited (see Fig.2). Also associated are two signals. The signal "nonempty" is sent each time a character is put into the buffer and implies the condition n > 0, and "nonfull" is sent each time a character is taken from the buffer and implies n < N, where N is the buffer's size..

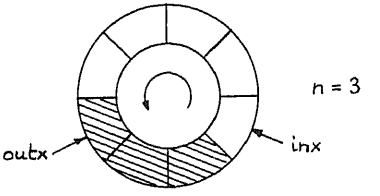

Fig.2, Cyclic buffer with 8 slots

The typewriter stream part is represented by the two device modules "keyboard" and "typewriter", each containing a buffer and a device driver, and of the main process "stream 1".

Each of the two buffers is accessed by a producer and a consumer process; n denotes the number of characters in the buffer.

```
Producer:  loop produce a character;
               if n = N then wait(nonfull) end;
               deposit the character in buffer;
               inc(n); send(nonempty)
           end

Consumer:  loop if n = 0 then wait(nonempty) end;
               fetch next character from buffer;
               dec(n); send(nonfull);
               consume the character
           end
```

The last three lines of the Producer and the first three of the Consumer are formulated as parts of a device interface module to ascertain mutual exclusion.

The important property of this solution is that no reference to machine and device dependent facilities is made outside the device modules, and that even within these modules the programs are straight-forward, referring to hardware defined facilities only in a few instances.
These are:
1. the device Status registers KBS and TWS,
2. the device Buffer registers KBB and TWB,
3. the device interrupt addresses 60B and 64B,
4. the interrupt priority 4.

Bit 6 in each status register is the interrupt enable bit. The statement xxS[6] := true (preceding "doio") enables, the statement xxS[6] := false (following "doio") disables interrupts.

Of course, in realistic applications the drivers are very short programs compared to the remainder of the system. Here the

opposite is true, because the most trivial case of a main process was chosen in order to be able to concentrate on demonstrating Modula's capability to express driver processes and device operations in a readable, "high-level" programming style (see program at the end of this chapter).

2.2. The card reader to line printer stream

The mechanism for the transmission of data from a card reader to a line printer is more complicated, although based on the same principle of buffering for each device. There exist four reasons for complications:

1. The line printer uses the 7-bit ASCII encoding of characters, whereas the card reader delivers a different encoding. We shall let the drivers accept and deliver data encoded as dictated by their devices, and let the main process perform the task of translation (see procedure convert).

2. The card reader, once activated, reads a full card, i.e. a portion of 80 characters. Hence, the buffer space reservation scheme must be somewhat different from the one used above.

3. If data transfer rates are high, it is inappropriate to exchange synchronization signals after each character transmission. Data are "blocked" and signals are sent only after transfer of each block. We demonstrate this solution in the case of the line printer, where each line is taken as a block.

4. The card reader status must be interrogated before and after reading each card. If it is in the "not ready" state, interrogation must be repeated periodically, because the card reader does not send a signal when returning to the ready state.

We notice that all four complications are not just due to the programmer's like for sophistication, but are caused by the inherent characteristics of the various devices and data carriers. We first describe the modifications to the input buffer scheme.

The circumstance that input data are delivered in portions of 80 characters requires that before starting the card reader, space for at least 80 characters must be free in the buffer. This entire portion must be reserved to the reader process during the time of reading the card. At the same time, the remainder of the filled buffer should be accessible to the consumer. Hence, a single variable n to indicate the number of filled elements does not suffice, since the number of empty portions is not necessarily N-n, when N is the total buffer size. In place of a single counter n, two variables nf and ne are introduced to denote the numbers of filled and empty slots respectively. The invariant

$$N-m1-m2 \leq ne+nf \leq N$$

holds at all times, where m1 and m2 are the portion sizes claimed by producer and consumer respectively. (Actually m1 = 81, since each card end is represented by a marker). In order to reduce the number of signalling operations, the "sleeping barber" scheme is used [2]. It is sketched by the following program excerpt:

```
Producer:  loop dec(ne,m1); if ne < 0 then wait(nonfull) end;
               deposit portion with size m1;
               inc(nf,m1) ; if nf ≥ 0 then send(nonempty)end
           end

Consumer:  loop dec(nf,m2); if nf < 0 then wait(nonempty) end;
               fetch portion with size m2;
               inc(ne,m2); if ne ≥ 0 then send(nonfull) end
           end
```

In the card reader driver the statement CRS := [0,6] sets the interrupt enable bit and at the same time initiates the reader motion. A completion interrupt is sent after receiving each column in CRB. This is represented by the statement doio in the usual way. It is followed by a test of bits 14 and 15, the former indicating that the end of the card is reached and a next card is in the hopper, the latter indicating that an "abnormal" condition exists, such as an empty hopper. Before starting a card motion, status bits 8 and 9 are tested. If bit 8 is set, the reader is not ready. It becomes ready when the operator pushes the RESET button. This action, however, does not send an interrupt signal to the computer. Hence, the status has to be polled repeatedly. Such a polling process is easily expressed by a statement of the form

```
while test fails do wait(s) end
```

if one can assume the availability of a signal s that is sent periodically at certain time intervals.

Such periodic signals are customarily produced by so-called clocks, i.e. devices that interrupt the processor at given intervals, usually that given by the line frequency. (50 or 60 cycles/sec). In the present program, the "clock" appears as a device like other peripherals. However, it does not have to be triggered into producing the next pulse; instead, the driver process merely needs to wait for the next pulse to occur. This waiting is expressed by the statement "doio" (which in this case is obviously a misnomer). As a consequence, the process merely consists of the statement

```
loop doio; send(tick) end
```

We are now tempted to substitute the clock signal (which is appropriately called "tick") for the signal s. However, this would violate the (peculiar) implementation restriction of Modula which specifies that no signal sent by a device process may be received by a device process. The difficulty is resolved by inserting yet another process - which is not a device process - between the sender and the receiver. This process is called

"clock". The clock driver process is enclosed in a device module (called "timing"), from which only the signal "tick" is exported. Note that Modula specifies that no process may execute a send operation upon an exported signal; hence, the device process is the only process that can send the tick signal.

The line printer buffering scheme is similar to that of the card reader in so far as signals are exchanged after receiving portions of data instead of after each character. In contrast to the card reader, however, the portions do not have a fixed size. We take the line as the natural choice of a portion, i.e. each end of a line, represented by a line-feed control character, causes a signal to be sent to the driver. Line printers usually have their own "hardware" buffer, in which characters are accumulated, until a line-feed character is received, whereupon the actual printing occurs. This explains the property that individual characters are accepted very quickly, i.e. instantaneously, because they are only stashed away in the internal buffer. As a consequence, it is unwise to wait for an interrupt from the printer in this case. Instead, the status register LPS is tested immediately after putting a character into the buffer register LPB. If Bit 7 is not set, printing takes place, and we may as well wait for the completion signal (interrupt) by executing "doio". (Note that printing is not caused exclusively by line-feed, but also as soon as the line buffer is filled. Therefore the test for the necessity of waiting must not depend on the value of the character output).

We presume, however, that the sender knows whether a special line-feed is sent (a line is closed) or an ordinary character is to be printed. The action of terminating a line is therefore expressed by an explicit procedure "weol" (write end of line), that is used to send a signal to the driver. The variable nf does not count the number of single characters in the buffer "buf", but instead counts the number of lines.

The main process "stream 2" merely receives characters from the card reader and transmits them to the line printer. Its function is to convert the character encodings. The line printer requires ASCII code, whereas the card reader delivers integers. Their binary representation is obtained in the way shown in Fig. 3 from the card hole combinations (this mapping is a hardware facility); the value x represents the position of a single punch in zones 1 to 7.

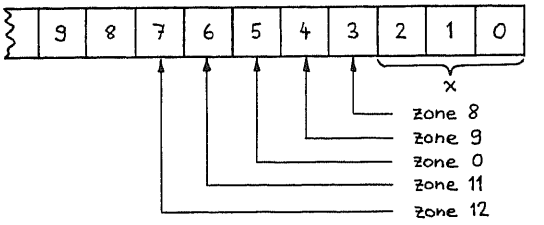

Fig.3, Card punch encoding

A special end-of-file card is recognised as one which has a 7-8-9 hole combination in column 1. It is translated into an integer value -1, and in turn to a form-feed character which causes a page eject on the line printer.

In order to show the convenience of the process and signal facilities, we shall introduce yet another complication that demonstrates a frequently occurring situation and originates from a very practical problem. We assume that the printer is a chain printer, and that the chain drive can be started and stopped under program control. We pose the condition that the chain motor is to be stopped, if no printing is requested for a period longer than a certain time period t. This is desirable to save the printer from undue wear. Naturally, the chain has to be restarted as soon as information has to be printed. As the starting of the chain is subject to a certain delay during which the printer is inoperable, the idle period t has to be chosen carefully. About 10 sec have proved to be a sensible value.

The "frequently occurring situation" referred to above is the condition in which a "process" is waiting for any one of several events. In our case it is the printer process having encountered an empty buffer, which is waiting for either new information to enter the buffer in order to be printed, or for the printer chain to be stopped after the time interval t has elapsed. This problem is solved in the following way: A new process is introduced with the purpose to control the printer chain movement. It is therefore called "chaincontrol", and consists of a simple loop containing two statements. The first of them, called "testchain", contains a wait statement, which "puts this process to sleep" while the printer is active. As soon as the printer driver recognises an empty buffer, it signals chaincontrol to become active. A delay counter "del" is initialised which determines the number of times chaincontrol executes its loop "testchain; wait(tick)". The second statement is merely a delay (20 msec) and demonstrates another use of the clock signal. If, during one of the executions of "testchain", it is discovered that the printer has resumed operation by finding nf $\geq$ 0, chaincontrol is immediately "put to sleep" again. If, on the other hand, the printer had remained inactive during the specified number of tests, an appropriate control character is output, causing the chain to stop, whereafter chaincontrol, having performed its task, waits for the next time

the printer needs to be guarded. The system consisting of the processes "chaincontrol" and printer driver assumes the following possible states:

	driver	chaincontrol	nf	del	chain
1	active	wait(guard)	≥Ø		running
2	wait(nonempty)	active	<Ø	>Ø	running
3	wait(nonempty)	wait(tick)	<Ø	>Ø	running
4	wait(nonempty)	wait(guard)	<Ø	=Ø	stopped
5	active	active	≥Ø		running

Transition 1 - 2 is caused by the driver sending the signal "guard", transitions 2 - 3 and 3 - 2 are implied by the execution of the control process, the latter by receiving the signal "tick". 3 - 4 occurs through the signal tick, and the control process issuing a dc4 character after decrementing del to Ø. Transition 4 - 1 occurs if new information enters the buffer, whereas 3 - 5 occurs for the same reason, but while the chain had not been stopped yet.

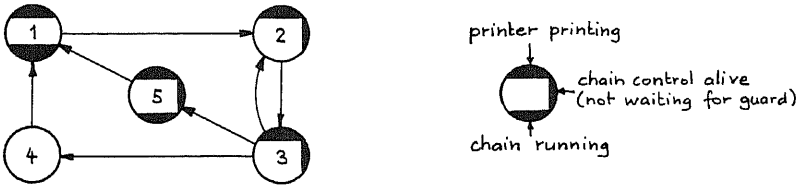

Fig.4, State transitions of printer system

Note that the procedure "testchain" is an interface procedure. Therefore it is guaranteed that during its execution the driver either waits (nf < Ø) or performs output operations (nf ≥ Ø).

We conclude the presentation of this program by a diagram showing its various processes and their interfaces represented by buffers and signals.

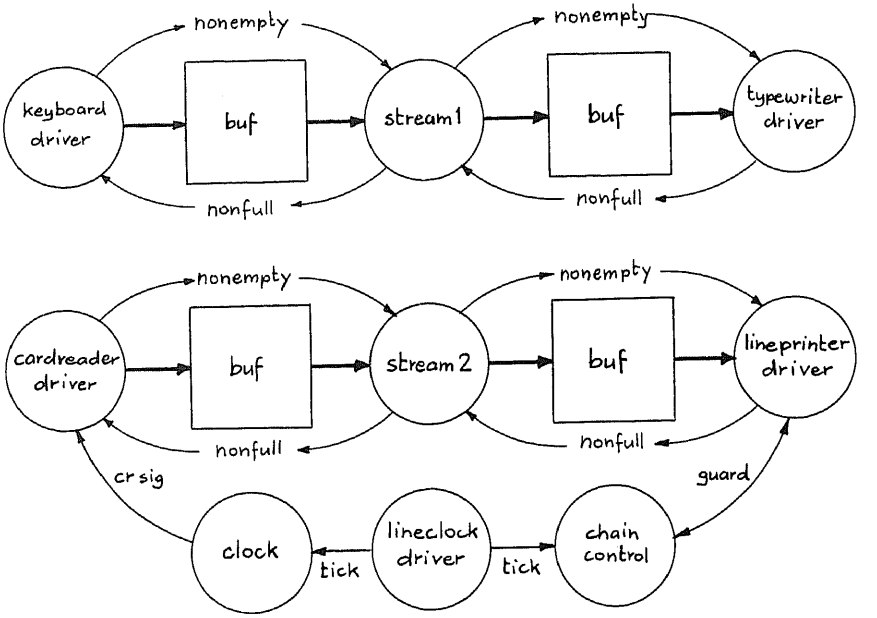

Fig.5, Interactions of processes via buffers and signals

```
module datastreams;
  const lf = 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
  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);
```

```
    begin
      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(0) 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 typewriter [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: array 1: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
  end driver ;

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

device module lineprinter [4];
  define write, weol, testchain;
  use lf;
  const n = 512;   (*buffer size*)
      dc3 = 23C; dc4 = 24C;
      chaindelay = 250;     (* 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);
      if del = 0 then
        lpb := dc4; wait(guard)
      end
    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
          lps[6] := true; doio; lps[6] := false
        end
      until ch = 1f;
      if ne >= 0 then send(nonfull) end
    end
  end driver ;

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

process stream1;  (*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 stream1 ;

process stream2;  (*cardreader to lineprinter*)
  use read, write, weol;
  const eoi = 378; badchar = '?';
  var x: integer; ch: char;
      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;
    zone := z[zone];
    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[ 3] := -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] := ':'; t[11] := '=';
  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] := ')'; t[29] := '\'; t[30] := '!'; t[31] := ';';
  t[32] := '-'; t[33] := 'J'; t[34] := 'K'; t[35] := 'L';
  t[36] := 'M'; t[37] := 'N'; t[38] := 'O'; t[39] := 'P';
  t[40] := 'Q'; t[41] := 'R'; t[42] := '+'; t[43] := '$';
  t[44] := '*'; t[45] := ']'; t[46] := '?'; t[47] := '>';
  t[48] := '0'; t[49] := '/'; t[50] := 'S'; t[51] := 'T';
  t[52] := 'U'; t[53] := 'V'; t[54] := 'W'; t[55] := 'X';
  t[56] := 'Y'; t[57] := 'Z'; t[58] := '['; t[59] := ',';
  t[60] := '('; t[61] := '_'; t[62] := '"'; t[63] := '&';

  loop read(x);
    if x = eoi 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
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*)
  stream1; stream2; clock; chaincontrol
end datastreams .
```

3. THE PROGRAM "DISKTAPE"

With the second program we continue and elaborate on the subject of device handling and sequential data transfer. The devices under consideration are a disk unit and a terminal containing two tape cassette drives. These devices are in several respects more complicated than those discussed in the preceding section, and therefore require also some more sophistication in the design of their operators. The equipment used in this particular experiment consists of a DEC RK-11 disk cartridge unit and an HP2644A terminal [3]. This terminal contains a microprocessor that is used to send and receive instructions to operate the two tape cassettes.

Let us first list the specifications of the program to be designed.

1. Let the disk store consist of 8 files with a fixed maximum length.
2. Commands are to be given from the terminal keyboard. There shall be commands for copying a file from tape to disk, and vice-versa, for rewinding, selecting, and marking tapes, and for skipping files on a tape.
3. Specifically, the commands are
   Rn: read tape onto disk file n ($1 \leq n \leq 8$).
   Wn: write disk file n onto tape.
   Cs: control operation; the character string s is determined by the specifications of the HP terminal.
4. Every command is to be terminated by a period. The system obeys the commands as soon as the "command line" is terminated (line-feed character). A line may contain several commands.
5. When typing a command, the entire command line can be cancelled by typing a backspace character.

The task thus defined appears as more difficult than the preceding problem for several reasons.

1. Instead of a number of continuous data streams, there is essentially one channel on which data are transferred according to input commands that must be interpreted.
2. The disk is a device which requires an address of the location where the data are stored.
3. The disk is a device that can read and write data.
4. The disk transfer rate is so high that we must avoid entering a device module for each character to be transferred.
5. The terminal is a device that requires a certain prescribed "protocol" to transfer data to and from the tapes. It also requires initiation and acknowledgement messages (also called "handshaking" procedures).

The overall structure of this system is therefore given by the devices involved. There are four processes: a terminal input, an output, a disk driver, and a main process controlling the transfers. The structure is shown in Fig.6, and the main process indefinitely repeats the following three statements:

```
loop read command;
     interpret command;
     acknowledge command
end
```

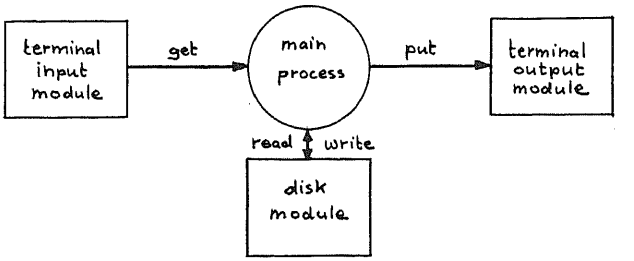

Fig.6. System structure with 3 device modules.

3.1. Module structure and interfaces

We start out by defining the interfaces between the modules, i.e. the procedures by which communication takes place. The terminal input module defines a procedure called get. It "gets" a character from the terminal. Similarly the output module defines a procedure put. The disk module, allowing transfers in both directions, defines procedures read and write. Naturally, these operations cannot be performed at the same time. The disk either reads or writes: its state is characterised by a mode variable. The disk module therefore defines another procedure, called open, to set this mode. Accordingly, a procedure close is defined to indicate termination of a file transfer, and to make the disk available for further operations.

The two modules which buffer the HP 2644A terminal are exactly like the ones used in the program of chapter 2. They implement a simple, cyclic buffer. Both buffers must be capable of holding at least one line of text (which is the unit of data transferred to and from tape at 2400 baud).

Before embarking on the explanation of the disk module, we explain the details of the main process. Because interpretation of a command shall begin only after the reading of an entire command line, and because commands arrive from the same source as the data to be transferred, a further buffering of commands is evidently necessary. This calls for another module, which we shall call scanner. It contains a buffer called line, and acts not unlike the scanner of a compiler, reading lines of text and delivering a command each time called upon. This command is a character sequence stored in the variable com. Note that the scanner also takes care of system specification 5 which demands that an input line can be cancelled by typing a backspace character. The variable n, appropriately hidden inside the scanner, counts the number of commands in the line. k is the index of the last character read from the command line. (See

program at the end of this chapter.)

3.2. Interpretation of commands

We can now proceed to an explanation of the detailed operations performed when interpreting a command. The first character (R,W,or C) identifies the command. In the cases of the R and W commands, the second character denotes the disk file to be written or read. In order to understand the data transfer mechanism, we must know the representation of text as received from and as required by the terminal. The essentials are as follows:

Data are read from and written onto tape by sending a "command" to the terminal. It consists of a sequence of characters starting with an escape character. The sequence is therefore called an escape sequence. The sequence

esc & p Ø R dc1

causes the terminal to read a line of text from the designated tape and to send it to the computer. The line is terminated by a cr and an lf character. (We assume that the HP 2644A terminal is strapped for line block transfer with straps D, G in, E, H out [4]). If the end of the file has been reached, a tape mark is encountered, which results in the transmission of a "line" consisting of the single control character rs (record separator), followed by cr, lf. Hence, it is always necessary to check the first character of each line received in order to detect the end of the file.

When writing on a tape, the escape sequence

esc & p W

is used. It causes the terminal to accept a sequence of data characters terminated by cr, lf, dc1. Since we shall store the data on disk in exactly the form as received from the terminal, we must again test the first character in each line. If it is rs, then a file mark has to be written instead of transmitting the "line". This is done by sending the escape sequence

esc & p 5 C dc1

After sending each line, an acknowledgement is requested from the terminal by sending a dc1 character. This acknowledgment consists of a letter S (success) or F (failure), followed by cr,lf. In our program, the acknowledgement is not interpreted; but it should obviously be done in a system acceptable for practical purposes.

The third command (C) is introduced merely to make available in a primitive manner the entire set of possibilities to control the terminal from the computer. The command C**s**, where **s** stands for any character sequence, is returned to the terminal as an escape sequence

esc & p s dc1

which may serve to designate source and destination tapes, to rewind tapes, to mark them, or for many other operations [4].

3.3. The disk module

Communication with the disk brings up two problems that have not been encountered when dealing with strictly sequential, unidirectional devices.

1. After transferring a file, the disk must be "released" in order to become available for the execution of another file transfer, whose direction may differ from the preceding one.

2. The high transfer rate of a disk requires that its interrupts be serviced with a high processor priority. It is desirable that the processor enters that priority level as rarely as possible. Communication with the device process must take place at that level, and should therefore be necessary only after transfer of relatively large blocks of data, and certainly not for each character.

The frequently adopted solution to problem 1 is that instructions (requests) for disk operations are queued in a buffer along with their parameter (disk address, buffer address, mode of transfer, etc.). We will refrain from request buffering, and show a solution that requires a much stricter synchronization of the two participating processes.

Assume that there are variables to hold the instruction parameters listed above. During an entire disk operation (file transfer) they belong exclusively to the disk driver process. Only after its termination and until the initiation of the next operation do they belong to the requestor (subsequently called "user"), which at this time may assign the parameters of the next request to them. This is an obvious situation where semaphores [2] are used to delay a process before passing a certain point, as is shown in Fig.7. We call the semaphore that delays the program before transmitting its next request "diskfree" (because the semaphore "opens" when the disk process is free to accept the next request), and we call the semaphore that delays the disk driver until the next request is deposited "userfree", because it opens when the user is free to work after having deposited the request.

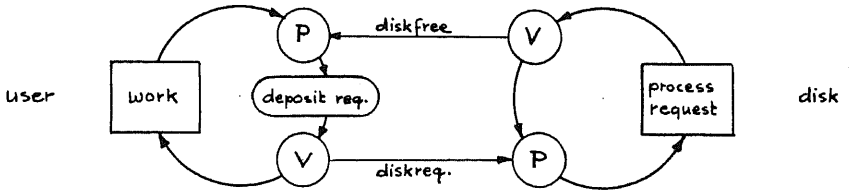

Fig.7. Requestor / acceptor synchronization with two semaphores.

According to Hoare [5], we express a (binary) semaphore by a pair consisting of a signal and a Boolean variable which

remembers whether a signal had been sent. The semaphore "diskfree" is represented by the signal of the same name and the Boolean variable "diskbusy" and the operations P and V become

```
P:  if diskbusy then wait(diskfree) end; diskbusy := true

V:  diskbusy := false; send(diskfree)
```

Similarly, the semaphore "userfree" is represented by the signal of the same name and the Boolean variable "userbusy". The two processes in Fig.7. are then expressed as follows:

```
user: loop if diskbusy then wait(diskfree) end;
           diskbusy := true;
           deposit disk request;
           userbusy := false; send(userfree);
           repeat produce/fetch data;
                  deposit/consume data
           until end of file
      end

disk: loop diskbusy := false; send(diskfree);
        (*now disk request parameters are loaded by "user"*)
        if userbusy then wait(userfree) end;
        userbusy := true;
        repeat fetch/read data;
               write/deposit data
        until end of file
      end
```

We notice a strong symmetry between the two processes, although in their roles they are dissimilar: the disk process is the slave of the user. In Modula, the disk process is formulated as a device process. The fact that device processes are served with a higher priority has in this case a notable consequence [7]: The send operation does not "release" the processor. Hence, no action can possibly intervene before the succeeding statement wait(userfree). We know that userbusy has the value true, because the only way it can obtain the value false is through the user process having received the signal diskfree. The consequence is that the variable userbusy can be deleted entirely, and the semaphore reduces to a signal, (which we rename "diskrequest").

Programming experience has shown that the mistake to use a simple signal where actually a semaphore would be needed, is one of the most frequent sources of system deadlocks. The reason is that a signal is only effective at the time it is sent; if no process is waiting for it, the signal is forgotten. The Boolean variable in the semaphore serves to remember that a signal was emitted (the general semaphore even counts them). The deadlock then arises when a process decides to wait for a signal that - incidentally - had been sent already when there was no one to receive it.

We now turn our attention to the second problem: avoiding to enter the driver module for fetching or depositing each single

character. We note that the disk reads and writes blocks of characters, so-called sectors with a fixed size of 512 characters (256 words). Hence, the solution lies in nesting the device module inside an interface module. The latter is entered for each character transmitted, the former for each sector.

We first consider the device module called disk. It contains, apart from the driver process, the three procedures opendisk, transmitsector, and closedisk, and the variables np, nd, diskstate, and the disk registers. The disk address register and the diskstate are initialised by the procedure opendisk. np and nd replace the usual counters nf and ne, and denote the numbers of sectors available to the programm (user) and the disk respectively. This change in nomenclature is essential, because the disk sometimes assumes the role of the producer, sometimes of the consumer. Now each of the two partners operates in a fixed way on its own counter, independently of the direction of data transfer. This is epitomized by the single procedure "transmitsector" replacing a readsector /writesector pair. (Note that np and nd do not include the sectors upon which the program or the disk are currently operating.)

Whereas the disk module views the buffer as a collection of sectors, the outer module, called diskio, views it as a buffer of single characters. In particular, it contains a counter variable nc counting the number of characters available for reading, or of empty slots available for writing in the sector currently belonging to the program. Once this count reaches zero, a new sector is transmitted (changes ownership). But only in this instance is the device module entered.

A few details may help the reader to comprehend the disk modules: A variable "mode" denotes the current state of the disk module (idle, reading, writing). It is used to set the value of the device module's analogous variable "diskstate", and to determine the actions needed upon closing a file. Moreover it could serve for checking the legality of calls of the read and write procedures.

The disk is initialised by setting the disk address register, the buffer address register, a word counter, and the control and command register. The latter requires setting of the following bits: 6 for interrupt enable, 2 for reading, 1 for writing, and 0 for starting the operation.

The disk driver needs to be able to recognise the last sector of a file to be read or written. A simple solution lies in embedding this information in the last sector itself. This is done in our example by giving the last character of the last sector the value fs (file separator), assuming that this character does not otherwise occur. In writing a file, this character is inserted by the procedure "close". It is recognised by the disk driver upon reading and writing. Note that the variable "diskstate" combines the functions of the Boolean "diskbusy" and of the driver's mode of operation.

The file system presented here is primitive with respect to disk

store allocation. It assumes a fixed allotment of 48 sectors (= 24575 characters) per file, because it was not the purpose of this exercise to develop a flexible storage allocator. The entire program is listed subsequently and is to be consulted for further details; the overall process- and module structure is shown in Fig.8.

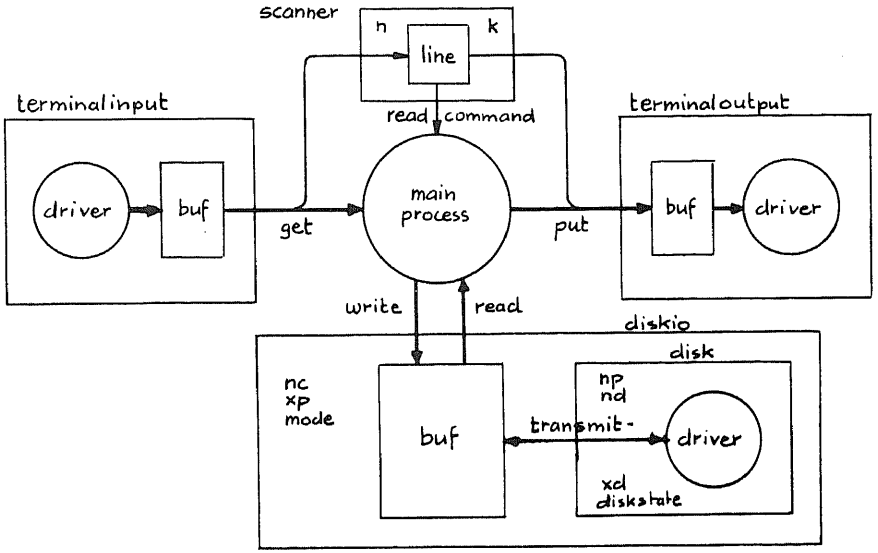

Fig.8. Module and process interconnections

```
module disktape;
const lf = 12C; cr = 15C; rs = 36C; dc1 = 21C;
  esc = 33C; bs = 10C; bel = 7C; fs = 34C;
var ch, fch: char;
  i: integer;
  com: array 1:80 of char;   (*current command*)

device module terminalinput [4];
  define get;
  use ch;
  const del = 177C; bufsize = 256;
  var n, in, out: integer;
      nonempty, nonfull: signal;
      buf: array 1:bufsize of char;

  procedure get;
  begin if n = 0 then wait(nonempty) end ;
    ch := buf[out]; out := (out mod bufsize) + 1;
    dec(n); send(nonfull)
  end get ;

  process driver [300B];
    var ts [175610B]: bits;
```

```
        tb [175612B]: integer;
        ch: char;
  begin
    loop if n = bufsize then wait(nonfull) end ;
      ts[6] := true; doio; ts[6] := false;
      ch := char(tb mod 200B);
      if ch = del then halt(0) end ;
      buf[in] := ch; in := (in mod bufsize) + 1;
      inc(n); send(nonempty)
    end
  end driver ;

begin n := 0; in := 1; out := 1; driver
end terminalinput ;

device module terminaloutput [4];
  define put;
  const bufsize = 256;
  var n, in, out: integer;
      nonempty, nonfull: signal;
      buf: array 1:bufsize of char;

  procedure put(x:char);
  begin if n = bufsize then wait(nonfull) end ;
    buf[in] := x; in := (in mod bufsize) + 1;
    inc(n); send(nonempty)
  end put ;

  process driver [304B];
    var ts [175614B]: bits;
        tb [175616B]: char;
  begin
    loop if n = 0 then wait(nonempty) end ;
      tb := buf[out]; out := (out mod bufsize) + 1;
      ts[6] := true; doio; ts[6] := false;
      dec(n); send(nonfull)
    end
  end driver ;

begin n := 0; in := 1; out := 1; driver
end terminaloutput ;

interface module diskio;
  define open, read, write, close;
  use ch,fs;
  const nsec = 4;          (*no. of sectors*)
    sectorsize = 512;
    bufsize = 2048;        (*bufsize = nsec*sectorsinze*)
    maxsectors = 48;       (*max no. of sectors per file*)
    nfl = 8;               (*no. of files*)
  var mode: integer;       (*0=free, 1=read, 2=write*)
    nc: integer;           (*no. of chars in current sector*)
    ns: integer;           (*no. of sectors written*)
    xp: integer;           (*buffer index of program*)
    current: integer;      (*index of current file*)
    file: array 1:nfl of
            record adr, size: integer
```

```
        end ;
  buf: array 1:bufsize of char;

device module disk [5];
  define opendisk, transmitsector, closedisk;
  use buf, bufsize, sectorsize, nsec, fs;
  var np, nd: integer;   (*no. of sectors available*)
    sp, sd: signal;
    diskstate: bits;
    diskreq, diskfree: signal;

    rkds [177400B]: bits;       (*drive status*)
    rker [177402B]: bits;       (*error status*)
    rkcs [177404B]: bits;       (*control status*)
    rkwc [177406B]: integer;    (*word count*)
    rkba [177410B]: integer;    (*buffer address*)
    rkda [177412B]: integer;    (*disk address*)

  procedure opendisk(m,a: integer);
  (*initialise disk in mode m at address a*)
  begin if diskstate[6] then wait(diskfree) end ;
    if m = 1 then
      diskstate := [0,2,6]; nd := nsec; np := 0
    else
      diskstate := [0,1,6]; nd := 0; np := nsec
    end ;
    rkda := a + 200000B; send(diskreq);
    dec(np); if np < 0 then wait(sp) end
  end opendisk ;

  procedure transmitsector;
  begin inc(nd); if nd >=0 then send(sd) end ;
        dec(np); if np < 0 then wait(sp) end
  end transmitsector ;

  procedure closedisk;
  begin inc(nd); if nd >=0 then send(sd) end
  end closedisk ;

  process driver [220B];
    const wps = 256;   (*wps = bufsize/2*)
    var endchar: char;
      xd: integer;     (*buffer index*)
  begin
    loop wait(diskreq); xd := 1;
      repeat dec(nd);
        if nd < 0 then wait(sd) end ;
        rkwc := -wps; rkba := adr(buf[xd]);
        rkcs := diskstate; doio; rkcs[6] := false;
        if rkcs[15] = 1 then halt(15) end ;
        inc(xd,sectorsize); endchar := buf[xd-1];
        if xd > bufsize then xd := 1 end ;
        inc(np); if np >= 0 then send(sp) end
      until endchar = fs;
      diskstate := []; send(diskfree)
    end
  end driver ;
```

```
  begin diskstate := []; driver
  end disk ;

  procedure open(m: integer; k: char);
  (*open file k in mode m*)
  begin mode := m; nc := sectorsize; ns := 0; xp := 1;
    current := integer(k) - integer('0');
    if (m = 1) and (file[current].size = 0) then halt(3) end ;
    opendisk(m, file[current].adr)
  end open ;

  procedure write;
  begin (*assume mode = 2, nc < sectorsize, ch <> fs*)
    buf[xp] := ch; inc(xp); dec(nc);
    if nc = 0 then
      if xp > bufsize then xp := 1 end ;
      transmitsector; nc := sectorsize;
      inc(ns); if ns = maxsectors then halt(1) end
    end
  end write ;

  procedure read;
  begin (*assume mode = 1*)
    if nc = 0 then
      if xp > bufsize then xp := 1 end ;
      transmitsector; nc := sectorsize
    end ;
    ch := buf[xp]; inc(xp); dec(nc)
  end read ;

  procedure close;
  begin (*assume mode = 1 or 2*)
    if mode = 2 then
      buf[xp+nc-1] := fs;
      inc(ns); file[current].size := ns
    end ;
    closedisk; mode := 0
  end close ;

begin (*interface module diskio*)
  mode := 0; current := nfl;
  repeat file[current].adr := current * 64;
         file[current].size := 0; dec(current)
  until current = 0
end diskio ;


module scanner;
  define readcommand;
  use ch, com, lf, bs, get, put;
  var k,n: integer;
    line: array 1:80 of char;

  procedure readline;
    var i: integer;
    (*assume linelength <= 80*)
```

```
  begin put('*'); i := 0; k := 0;
    repeat get;
      if ch = bs then  (*cancel*)
        put('\'); i := 0; n := 0;
      else
        inc(i); line[i] := ch; put(ch);
        if ch = '.' then inc(n) end
      end
    until ch = lf
  end readline ;

  procedure readcommand;
    var j: integer;
  begin
    while n = 0 do readline end ;
    dec(n); j := 0;
    repeat inc(j); inc(k); com[j] := line[k]
    until line[k] = '.'
  end readcommand ;

begin n := 0
end scanner ;

procedure escseq;
begin put(esc); put('&'); put('p')
end escseq ;

procedure acknowledge;
begin put(dc1); get;
  if ch <> 'S' then put(ch); put(bel) end ;
  repeat get until ch = lf
end acknowledge ;

begin (*main*)
  loop readcommand;
    if com[1] = 'C' then          (*control*)
      escseq; i := 2;
      while com[i] <> '.' do
        put(com[i]); inc(i)
      end ;
      acknowledge
    elsif com[1] = 'R' then       (*read tape*)
      open(2,com[2]);
      repeat (*read a line*)
        escseq; put('0'); put('R'); put(dc1);
        get; fch := ch; write;
        repeat get; write
        until ch = lf
      until fch = rs;
      close
    elsif com[1] = 'W' then       (*write tape*)
      open(1,com[2]); read;
      while ch <> rs do
        escseq; put('W'); put(ch);
        repeat read; put(ch)
        until ch = lf;
        acknowledge; read
```

```
      end ;
      escseq; put('5'); put('C'); (*mark tape*)
      acknowledge; close
    end ;
    put(bel)
  end
end disktape .
```

4. THE PROGRAM "SPACERACE"

This program represents a system containing m identical processes which may be thought as being "vehicles" racing around on a plane according to their own laws (algorithms) of motion. Each of them is characterised by a position on that plane, represented by Cartesian coordinates, and a velocity. Another process drives the display device which shows the plane, i.e. its boundaries, and the m vehicles.

In order that these vehicles move on the screen "in real time", their coordinates are updated in discrete intervals of time called the update interval. We let this interval be (a multiple of) the clock pulse interval defined by the computer's line clock device (20 ms). A vehicle is thus described by a process of the form

```
loop wait(tick); update coordinates end
```

This program differs from the previous program in several significant aspects:

1. There are many replicas (instances) of the same process pattern.

2. The interaction between the vehicles and the display driver is quite strong (all coordinates are shared interface variables), but it does not require mutual exclusion: the (refresh) display driver may easily access coordinates at the same time as a vehicle is updating them.

3. It is convenient to represent the coordinates x and y of vehicle i ($1 \leq i \leq m$) as elements of an array. Whereas each vehicle process i accesses x[i] and y[i] only, the display driver accesses x[k] and y[k] for all k. It is therefore impossible to represent these coordinates as simple variables of a distinct interface module. Fortunately, it is also unnecessary.

Let us first define the process which constitutes the behaviour of a vehicle. Each process is characterised by a position (x,y) and a velocity (dx,dy) ; dx is the amount by which x is incremented after each (real) time interval d dictated by the internal clock. In the same way dy is the increment of y . As soon as a vehicle hits the space boundary - we assume that the area is a rectangle adapted to the display screen - it bounces like a ball. This is represented by a sign inversion of the

appropriate velocity component. Let the elapsing of a time interval t be signalled by a tick, then the process may be formulated as follows:

```
process vehicle (i: integer);
begin (*assume x, y, dx, dy are initialised*)
     loop wait(tick);
          inc(x, dx);
          if "x outside boundaries" then
             dx := -dx; inc(x, dx) (*bounce*)
          end;
          inc(y, dy);
          if "y outside boundaries" then
             dy := -dy; inc(y, dy) (*bounce*)
          end
     end
end vehicle
```

The structure of the display driver (called "screen") is again a loop, expressing the repeated action of drawing the same picture (refreshing). We assume that the delay between two drawings is again given by the tick signal of the real time clock (it must be no more than 40 ms, because otherwise the screen starts to flicker).

```
loop wait(tick); i := m;
     repeat display picture of vehicle i; dec(i)
     until i = 0
end
```

The details of the statement "display picture of vehicle i" are determined by the peculiarities of the display processor GT 44. Explanation of the most important principles may suffice at this point; for further details the reader is referred to the manufacturer's manuals [6].

1. The display processor is initiated by loading a storage address into the display program counter register (DPC). The processor then starts interpreting a sequence of instructions stored in consecutive locations starting at the given address.

2. There are so-called mode instructions which bring the processor into one of several modes for drawing lines, displaying text, or moving the beam to specified coordinates. Such coordinates are interspersed as data within the mode instructions, and are interpreted according to the set mode. (These data are absolute, there is no indexing. Hence the display processor is programmed like a very old-fashioned computer without index registers, conditional jumps, or subroutines. Our program exhibits this sorry state by listing display commands as octal numbers!)

3. There exists a halt instruction which causes the main processor to be interrupted.

From these explanations it follows that a display code sequence

must be set up that defines the picture by which a vehicle shows up on the screen. We choose this to be a small square box enclosing a letter which identifies the vehicle (A ... H for m = 8), but shall not further describe the details of the display code sequences. It is important to note, however, that the coordinates of each vehicle must be inserted in the code before the display processor is initiated. Hence the statement "display picture of vehicle i" is expressed (refined) as follows:

```
fig[c1] := x[i]; fig[c2] := y[i];
fig[c3] := code for letter i;
DPC := adr(fig); doio
```

The "system" developed so far is not a very exciting one, because all vehicles move with fixed speed and fixed direction, which is only changed when the vehicle bounces at a screen (frame) boundary. We shall therefore seize the opportunity to introduce one more complication: speed and direction of the vehicle movements are to be made variable. Specifically, they shall be influenced by the interactive use of a light pen.

For this purpose, we postulate that n "commands" are to be displayed on the screen in addition to the vehicle (and the frame) themselves. Let us postulate the following n = 8 commands:

1. increase speed in the x-direction.
2. increase speed in the y-direction.
3. decrease speed in the x-direction.
4. decrease speed in the y-direction.
5. increase the update-interval (del)
6. decrease the update-interval
7. reverse the direction of movement
8. halt the vehicle at its current position.

We represent these commands by appropriate pictures in a (reserved) command area of the screen (see Fig 8).

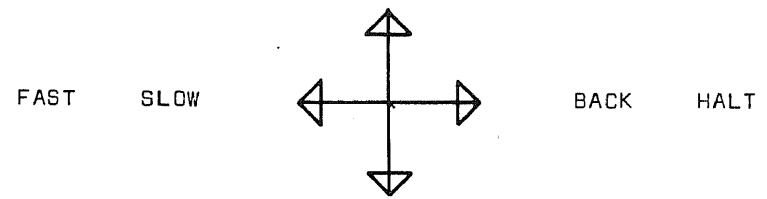

Fig.8, Commands as displayed on screen

Of course, pointing to a command must affect a single vehicle only. Which one? We solve this identification problem by postulating that at all times one of the m vehicles be the "designated vehicle". It is designated by being spotted by the light pen. (Moreover, we shall ask that the letter identifying the designated vehicle be blinking. This is easily accomplished by an appropriate display instruction).

These considerations lead to the introduction of a variable z denoting the index of the designated vehicle. But how do we introduce the notion of the light pen seeing an item into our program? The solution is remarkably simple: the pen is represented by a pen driver process (called "pen"). The process waits for the pen catching sight by executing "doio". Therefore this process has once more a loop structure:

```
loop doio;
     identify object seen and act accordingly
end
```

But how is the object seen identified? The hardware provides registers which hold the coordinates of the point being seen. But this facility is rather unhelpful, because it requires an elaborate back-translation of coordinates to point or line or figure being identified. Therefore we ignore this facility and choose a much more effective solution. The identity of the currently drawn item (frame, figure, or command) is set and retained by the screen process in the form of two variables. "mode" tells whether the frame, a figure, or a command is written (and therefore seen), and i tells which vehicle or command. These two variables therefore belong to the interface between the device processes "screen" and "pen", which are declared within the same device module. The first action after the pen has seen an object, is to determine the current mode. If it is "figure", the appropriate action is "z := i", because the seen vehicle becomes the designated vehicle. If the mode is "command", then the designated vehicle's speed dx[z] or dy[z] or its update interval del[z] have to be changed appropriately (see program).

An inherent problem in the use of a light pen is that it has to be disabled temporarily after having spotted an item, because otherwise during the next refresh cycle it will see the same item again. The disabling must last an appropriate time to allow the withdrawal of the pen from the screen, whereupon it has to be reenabled. In the DEC GT-44 system, the pen is not disabled; instead, the entire display is stopped. It is up to the pen process to restart it. This action must include the explicit execution of a display processor instruction that disables interrupts due to the pen, because there is no status register with an interrupt enable bit. (In our example, this instruction is included in the frame code.) This restart action of the screen within the pen process reveals that the current hardware is inappropriate to this method of structuring processes. The display device deviates in several respects unnecessarily from the comparatively regular and systematic handling of other devices, and in particular the light pen appears like an afterthought to the overall design. The present solution can at best be called an "elegant paint job", but it cannot hide the inherent incompatibility between the offered hardware and the systematic method of identifying devices as processes. Having executed the appropriate actions to interpret the seen command, and after having restarted the display processor, the pen process waits for an appropriate number of tick intervals, until it reenables the pen interrupts. The pen rests inoperative for

about 0.5 sec.

Fig. 9 displays the resulting process structure of the entire system, and lists the respective interface variables and signals.

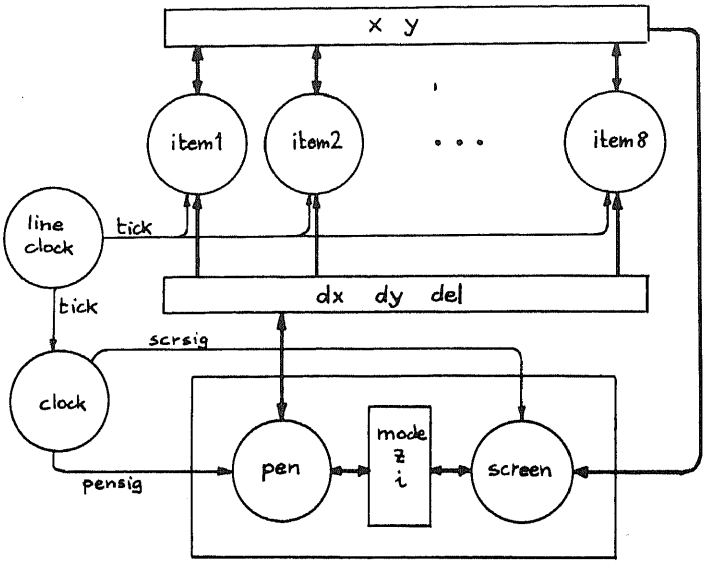

Fig.9. Processes, signals and shared variables

```
module spacerace;
  const m = 8;    (*no. of vehicles on screen*)
  var i,x0,y0: integer;
      scrsig, pensig: signal;
      x, y:    array 1:m of integer;    (*coordinates*)
      dx, dy: array 1:m of integer;    (*increments*)
      del:     array 1:m of integer;    (*delay steps*)

device module display [4];
  use x, y, dx, dy, del, pensig, scrsig, m;
  const n = 8;    (*no. of commands*)
  var mode: (frame, figure, command);
      i: integer;
      z: integer;  (*index of currently identified vehicle*)
      dpc [172000B]: integer;    (*display program counter*)
      frm: array 0:17 of integer;    (*frame display code*)
      fig: array 0:13 of integer;    (*vehicle display code*)
      com: array 1:n, 0:15 of integer; (*command codes*)

  process screen [320B];
  begin
  loop wait(scrsig);
    mode := frame; dpc := adr(frm); doio;
```

```
      mode := figure; i := m;
      repeat fig[1] := x[i]; fig[2] := y[i];
        fig[11] := 64 + i;
        if i = z then fig[10] := 103630B   (*blink on*)
                 else fig[10] := 103620B   (*blink off*)
        end ;
        dpc := adr(fig); doio; dec(i)
      until i = 0;
      mode := command; i := n;
      repeat dpc := adr(com[i]); doio; dec(i)
      until i = 0
    end
  end screen ;

  process pen [324B];
     var d: integer;
  begin
    loop frm[0] := 117164B; doio;   (*wait until pen signals*)
         frm[0] := 117125B;
      if mode = figure then z := i
      elsif mode = command then
        case i of
          1: begin inc(dx[z]) end ;
          2: begin inc(dy[z]) end ;
          3: begin dec(dx[z]) end ;
          4: begin dec(dy[z]) end ;
          5: begin inc(del[z]) end ;
          6: begin dec(del[z]) end ;
          7: begin dx[z] := 0; dy[z] := 0; del[z] := 0;
               if z = 1 then halt(0) end
             end ;
          8: begin dx[z] := -dx[z]; dy[z] := -dy[z] end ;
        end
      end ;
      dpc := adr(frm);   (*restart display*)
      d := 25;
      repeat wait(pensig); dec(d) until d = 0
    end
  end pen ;

begin (*display module; see note following this program*)
  frm[ 0] := 117124B; frm[ 1] :=      0B; frm[ 2] :=      0B;
  frm[ 3] := 110000B; frm[ 4] := 041777B; frm[ 5] :=      0B;
  frm[ 6] := 040000B; frm[ 7] := 001777B; frm[ 8] := 061777B;
  frm[ 9] :=      0B; frm[10] := 040000B; frm[11] := 021777B;
  frm[12] :=      0B; frm[13] := 000200B; frm[14] := 041777B;
  frm[15] :=      0B; frm[16] := 173400B;
  fig[ 0] := 117024B; fig[ 1] :=      0B; fig[ 2] :=      0B;
  fig[ 3] := 104000B; fig[ 4] := 050000B; fig[ 5] := 040040B;
  fig[ 6] := 070000B; fig[ 7] := 040140B; fig[ 8] := 130000B;
  fig[ 9] := 002206B; fig[10] := 103620B; fig[11] :=      0B;
  fig[12] := 173400B;
  com[1,0] := 117024B; com[1,1] := 001000B; com[1,2] := 000100B;
  com[1,3] := 104000B; com[1,4] := 056000B; com[1,5] := 062010B;
  com[1,6] := 040120B; com[1,7] := 042010B; com[1,8] := 173400B;
  com[2,0] := 117024B; com[2,1] := 001000B; com[2,2] := 000100B;
  com[2,3] := 104000B; com[2,4] := 040070B; com[2,5] := 062110B;
```

```
    com[2,6] := 044000B; com[2,7] := 062010B; com[2,8] := 173400B;
    com[3,0] := 117024B; com[3,1] := 001000B; com[3,2] := 000100B;
    com[3,3] := 104000B; com[3,4] := 076000B; com[3,5] := 042010B;
    com[3,6] := 040120B; com[3,7] := 062010B; com[3,8] := 173400B;
    com[4,0] := 117024B; com[4,1] := 001000B; com[4,2] := 000100B;
    com[4,3] := 104000B; com[4,4] := 040170B; com[4,5] := 062010B;
    com[4,6] := 044000B; com[4,7] := 062110B; com[4,8] := 173400B;
    com[5,0] := 117620B; com[5,1] :=    300B; com[5,2] :=     70B;
    com[5,3] := 100000B; com[5,4] := 046123B; com[5,5] := 053517B;
    com[5,6] := 173400B;
    com[6,0] := 117620B; com[6,1] :=    500B; com[6,2] :=     70B;
    com[6,3] := 100000B; com[6,4] := 040506B; com[6,5] := 052123B;
    com[6,6] := 173400B;
    com[7,0] := 117620B; com[7,1] :=   1200B; com[7,2] :=     70B;
    com[7,3] := 100000B; com[7,4] := 040510B; com[7,5] := 052114B;
    com[7,6] := 173400B;
    com[8,0] := 117620B; com[8,1] :=   1500B; com[8,2] :=     70B;
    com[8,3] := 100000B; com[8,4] := 040502B; com[8,5] := 045503B;
    com[8,6] := 173400B;
    mode := frame; z := 1; screen; pen
end display ;

device module timing [6];
    define tick;
    var tick: signal;
        lcs [177546B]: bits;

    process clock [100B];
    begin lcs[6] := true;
        loop doio;
          while awaited(tick) do send(tick) end
        end
    end clock;

begin clock
end timing ;

process vehicle(i: integer);
  var d: integer;
begin
  loop d := del[i];
    repeat wait(tick); dec(d)
    until d < 0;
    inc(x[i], dx[i]);
    if (x[i] < 0) or (x[i] > 1740B) then
      dx[i] := -dx[i]; inc(x[i], dx[i])
    end ;
    inc(y[i], dy[i]);
    if (y[i] < 200B) or (y[i] > 1740B) then
      dy[i] := -dy[i]; inc(y[i], dy[i])
    end
  end
end item ;

process clock;
begin
  loop wait(tick); send(scrsig); send(pensig)
```

```
  end
end clock ;

begin (*spacerace*)  i := m;
  xØ := 2ØØØB/(m+1); yØ := 22ØB;
  repeat x[i] := i*xØ; y[i] := yØ;
    dx[i] := Ø; dy[i] := Ø; del[i] := Ø;
    vehicle(i); dec(i)
  until i = Ø;
  clock
end spacerace .
```

The arrays frm, fig, and com represent the code for the GT-44 display processor. Its "language" is not part of Modula, and the commands are therefore denoted by octal numbers [6]. The following alternative specifications (for frm and fig only) should convey an idea of the nature of this code.

```
picture frm:  (*frame*)
begin point[intensity 4, blink off, solid lines](Ø,Ø);
  longvector(L,Ø)(Ø,L)(-L,Ø)(Ø,-L)(Ø,128)(L,Ø)
end frm;

picture fig(x,y: integer; z: char);  (*vehicle*)
begin point[intensity 4, blink off, solid lines](x,y);
  shortvector(D,Ø)(Ø,D)(-D,Ø)(Ø,-D);
  relativepoint(9,6);
  character[intensity 7, blink on] z
end fig
```

L = 1Ø23 is the frame width, and D = 32 is the width of the square depicting a vehicle.

References

1. N.Wirth, "Modula: A language for modular multiprogramming", Institut für Informatik, Report 18, ETH Zürich, March 1976.
2. E.W.Dijkstra, "Cooperating sequential processes", in Programming Languages, F.Genuys, ed., Acad. Press, London 1968.
3. R.G.Nordman, R.L.Smith, L.A.Witkin, "New CRT terminal has magnetic tape storage for expanded capability", Hewlett-Packard Journal 27, 9, 2-15 (May 1976).
4. 2644A Mini Data Station, owner's manual, Hewlett Packard manual No. Ø2644-9ØØØ1 (Nov.1975).
5. C.A.R.Hoare, "Monitors: An operating system structuring concept", Comm.ACM 17, 1Ø, 549-557 (Oct.1974).
6. GT-44 user's guide. Digital Equipment Corp. DEC-11-HGT44-A-D.
7. N.Wirth, "Design and implementation of Modula", in this Report.

DESIGN AND IMPLEMENTATION OF MODULA

N.Wirth

Abstract

This paper gives an account of some design decisions made during the development of the programming language Modula. It explains the essential characteristics of its implementation on the PDP-11 computer, in particular its run-time administration of processes and the mechanism of signalling. The paper ends with some comments on the suitability of the PDP-11 for this high-level multiprogramming language.

Author's address:
Institut für Informatik, ETH, CH-8092 Zürich

DESIGN AND IMPLEMENTATION OF MODULA

1. Introduction

In this report I shall try to do two things: to give an account of how and why certain design decisions were made in the development of the programming language Modula, and to explain some details of its implementation on the PDP-11 computer. The reader is referred to the defining report of Modula [1] which also contains a brief overview of the language, and to the companion paper illustrating some typical applications of Modula [2].

The entire project started out with the modest aim to gain some experience and insight in the field of multiprogramming and device handling. Such insight, it was stipulated, could well lead to a set of practical rules or guidelines for effective and reliable multiprogram system design, that could ultimately consolidate into a discipline. It soon became evident that the creation of such a set of rules amounted to nothing less than the design of a new "language", i.e. a set of notations and structures in terms of which design would proceed. Some programs were developed in this way and formulated in assembly code. This approach made it possible to experiment whith some proposed elements on which the design discipline was to evolve, but more significantly it also drove home a message that actually should have been known already: that the mere proposal of an abstract notational scheme to design systems (which thereafter must be coded by hand in a low-level language) will neither lead to reliable products nor to a significant saving in development cost. The necessity of a high-level language with a fully automated compilation process is unquestionable. The modest learning effort in multiprogramming thus slowly evolved into another exercise in language design and implementation.

Our desire to take over many familiar elements from Pascal is understandable. After all, our aim was to concentrate on the novel aspects due to multiprogramming, and not on the design of an entirely new language in toto. Yet, the occasion was seized to change several details, and in particular to omit certain facilities that are either complicated to implement, or not needed in the context of our original goal, or both. In retrospect, Modula turned out to be a language considerably smaller than Pascal. Yet it provides ample room for experimenting and exploring good techniques of programming, for committing blunders and recognising that multiprogramming is truly difficult even with a neat and systematic tool at hand.

The development of the compiler was started when a sketchy definition of the language had been laid out. The compiler was written in Pascal as a one-pass cross-compiler running on the CDC 6400. This approach allowed a relatively quick adaptation to language specification changes, which were rather frequent for a considerable time. Yet the exercise proved once again that, unless a very large part of the language is "right" at the beginning, such a project has a small chance to survive. It is

particularly important to choose the right moment when a language definition is to be stabilized, i.e. when a report is to be released for programmers, and when a second effort is to be launched aiming at a well-engineered, efficient compiler.

This paper discusses a few selected topics only, in particular those that distinguish Modula from Pascal. These are the module structure and all facilities concerned with concurrent processes. The discussion of aspects of implementation is restricted to the run-time organization; we refrain from commenting on compiler technology.

2. Modules

The block structure of Algol and Pascal with its facility to declare local objects does not adequately cover the needs of systems programming. In particular, it does not allow to hide objects (or details about them), while they are still in existence. Objects cease to exist, as soon as control leaves the procedure (block) to which they are local. To some degree, the own variables of Algol 60 incorporate the desired information hiding property. However, the own concept has well-known deficiencies, and a different solution has to be found.

A very much more appropriate solution was offered by the class concept of Simula as modified by Brinch Hansen and Hoare [3,4,5]. The class definition defines a set of procedures (operators) and a set of variables to which only these procedures have access. The important aspect is that these variables continue to exist, if control leaves any of these procedures.

Why then, did we not adopt the class structure in Modula? The primary reason is that the unit of program which has to encapsulate information from its environment - now called a module - appears to be of relatively large size in systems programming, and that usually only one instance of such a module exists. This is in contrast to the original aim of the class concept, where there exist many such objects of identical structure. They form a class . The class embodies the idea of an abstract data type to be declared in conjunction with its applicable operators [6,7]. The module, however, rather pursues the aim of an adequate facility to declare such entities as, for example, a scanner in a compiler, a disk store manager in an operating system, or a communication line handler in a data station. Hence, the module has somewhat different objectives than the class, and it would therefore be misleading to claim that the module concept replaces the class concept, although it can be shown that formally the former covers the latter. We shall show how a class definition can be represented by a module declaration, and then discuss the advantages and disadvantages of the two structures.

Consider the following class definition [4]:

```
type C = class
           var x,y: T;
           procedure p(U);
               begin ... end;
           procedure entry q(V);
               begin ... end;
         begin S
         end
```

This is expressed in terms of a module as follows:

```
module M;
   define R,q,s;
   type R = record x,y: T end;
   procedure p(var r: R; U);
      begin ... end p;
   procedure q(var r: R; V);
      begin ... end q;
   procedure s(var r: R);
      begin S
      end;
end M
```

The advantage of the class is evident, if we create several instances of the data structure, each consisting of the two components x and y:

```
var a,b: C
```

Thereby the initialization statement S is implicitly invoked once for a and once for b. An invokation of procedure q, applied to a is expressed conveniently as

```
a.q(v)
```

where a appears as a parameter in a distinguished position. In the case of the module structure, the two variables are declared as

```
var a,b: R
```

and the call as

```
q(a,v)
```

Initialization must be stated explicitly as s(a). The principal advantage of the class notation appears, if there are several classes, perhaps with the same identifiers for some of their individual operators: their names are entirely local, and their identity is determined by the prefixed parameter, whose type uniquely specifies one class.

It is not our intention to belittle these advantages, but the sophisticated combination of using a distinguished parameter position to identify the scope in which its operator is defined

is of much lesser importance in the application area envisaged for modules. It seems only natural that the designer of a system chooses unique names for all operators exported from different (parallel, not nested) modules. The primary advantage is then one of conceptual simplification: the module has one and only one function, namely to establish a static scope of identifiers, whose acrossboundary visibility of identifiers is strictly under the programmer's control. Export (and import) rules are not restricted to some operators (procedures), but apply identically to names of any kind of object, such as constants, types, variables. This generality of scope rules has in practice proved to be highly valuable.

If only one instance of a class variable is to appear, then of course the module notation is equally simple, if not simpler. The declaration

```
module M;
   define q;
   var x,y: T;
   procedure q(V);

        ....
begin S
end M
```

replaces both the class definition and its instantiation a:C, and the call q(v) replaces a.q(v). In the case of multiple instances the module notation is more cumbersome, but also provides some additional possibilities, such as, for example, the introduction of variables common to all class instance operators declared in the module heading and initalised in its body.

Implementation of the module is, considering the additional possibilities, simpler than that of the class. After all, the only conceptual innovation of the module lies in delimiting the scope of identifiers. In the compiled code, there appears no trace of the module structure itself. Nevertheless, the additional sophistication of the compiler's symbol table mechanism is rather considerable, and in any case greater than anticipated. It weighs particularly in the case of multipass compilers. Unfortunately, it appears to be almost impossible to avoid a multi-pass scheme. This not only in view of Modula's application to minicomputers with limited store size, but primarily because of the potential desirability of cross references of objects defined in different modules.

There are two issues that gave rise to some discussions: the read-only nature of exported variables, and the intransparency of exported types. The latter stems from the desire to match the class definition's power of internal structure hiding. Hence, a programmer using an exported type (e.g. R in the above example) cannot use any knowledge about the details of R, not even does he "know", for example whether R is a record or an array structure.

The decision to declare every exported variable to be a read-only variable is based on the view that variables belonging to a module actually should not be exported at all. Making them exportable as read-only variables, however, avoids the cumbersome declaration of function procedures that merely yield a variable's value.

3. General Processes

The most important and predominant decisions to be taken in the design of a multiprogramming language implementation are those concerning storage allocation and processor management (scheduling). The guiding criteria in the decisions taken here were efficiency and simplicity of addressing (of variables) and of signalling operations. The latter are the only programmed operators that may cause the processor to switch from one process to another. Another objective was a minimal run-time support routine.

3.1. Storage layout

The language Modula projects a view of systems consisting of a fixed number of concurrently active processes. This concept eliminates the need for a dynamic storage management scheme among processes. But even for primitive systems, this view of an entirely static world is somewhat too simplicistic: even simple systems, upon "deadstart", grow from an initial nucleus to their "operating" complexity. Modula therefore supports the possibility of dynamic process generation (and allocation of their workspace). However, it does not support the notion of their dissolution. (In fact, a process may terminate, but this does not imply the recycling of its store). Modula restricts the ability to generate new processes to the main program part. This brings two advantages. First, all complications arising from processes having "sons" (which perhaps survive their ancestors) are avoided. Second, after the main program has reached a specific point (usually the end), it is guaranteed that no storage overflow can occur. This is crucial in all process control systems, where the main program part assumes the role of an initialization phase. During this phase, storage is allocated sequentially as needed by newly generated processes. Each data segment contains a header containing a link to another process. All processes are thus linked in a single ring. This header, called the process descriptor also represents the state of the process when it is not being served by the processor.

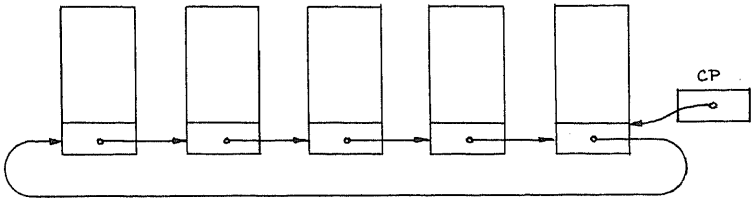

Fig.1. Workspaces of processes linked in a ring.

Naturally, each process has its own stack (work space). Each stack is of fixed size. This size may be computed by the compiler, provided that no procedure is activated recursively. Modula, however, does allow recursion. In this case, a compiler directive must indicate the maximum depth of the recursion.

3.2. Addressing of variables

A dominant problem is the addressing of variables, and the efficiency of the overall implementation crucially depends on the quality of its solution. Global variables (i.e. those declared in the main program block) can be addressed directly, as their location is known at compile time. Variables local to a process must be addressed relative to that data segment's origin. Since such references occur frequently, it is important that their access is efficient. Therefore, the origin of the segment of the currently running process is held in a processor register (called CP for Current Process).

Variables local to a procedure must be addressed relative to that procedure's data segment (also called activation record). The common solution is to stack these segments in the workspace (stack of the process), and to access segments via a descending link chain originating from a register pointing to the topmost segment. This solution guarantees most efficient access to local variables and less efficient access to variables local to surrounding procedures. As such accesses are much less frequent, this kind of sacrifice seems tolerable. If it is not, the mechanism of a Display can be introduced which, however, incurs some overhead upon procedure calls. The chief drawback of the described method is that a special register is required along with instructions to change its value upon each procedure call and exit.

The PDP-11 subroutine call mechanism uses a stack address register, and it happens that this register may also be used as a relative address for accessing local variables. The compiler is able to determine its value at each point, and hence also the offset of any variable. This technique makes variable addressing, even with the use of recursion, very simple and efficient. But there is, alas, again a hitch: If a program fails, then it is virtually impossible to generate a symbolic post-mortem-dump without the presence of a register containing the base address

of the most recently called procedure. If, due to these considerations, we reserve not only a register CP (current process), but also a register PP (procedure pointer), we obtain the following addressing modes:

1. Global variables:	direct
2. Variables local to a process:	relative to CP
3. Variables local to procedure p	
a. accessed within p:	relative to PP
b. accessed from procedures local to p:	indirect via PP

3.3. Operations on signals

A process can be in any one of three distinct states. It can be under execution by the processor (or a processor); then it is called running. Or it can be waiting to be executed (resumed); then it is called ready. Or it can be waiting for a signal s to be sent; it is called waiting on s. If Modula is to be implemented on a single-processor computer - and we henceforth presume this to be the case - we need not distinguish between the former two as far as state representation by the descriptor is concerned. If a process is waiting on s, its descriptor is linked into the queue of descriptors originating at variable s. It follows that signals are implemented very simply as variables with pointer values. (see Fig.2).

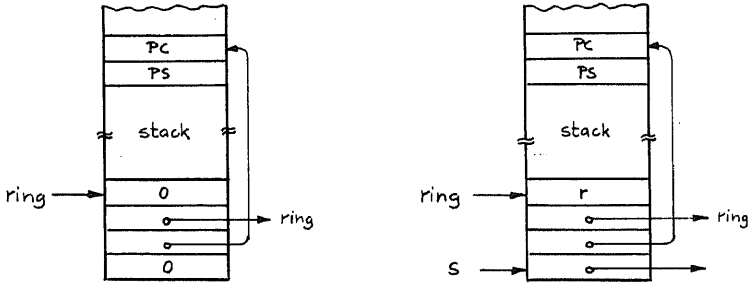

Fig.2. Process descriptors representing ready and waiting processes respectively.

Both in the ready and waiting states, the descriptor represents the state of the process. (Note: PC (program counter) and PS (processor status) are not deposited in the descriptor itself, but instead on the top of the (current) stack. The descriptor contains, however, the stack pointer SP). The operations of sending and waiting for a signal s are now quite straightforward.

Send(s): The PC and PS are dumped on the stack (e.g. by a trap instruction). Then SP is stored in the descriptor, the status field is set to Ø, indicating "ready". Then the first descriptor in the queue s is delinked, and CP is made equal to s. At last, SP, PC and PS are loaded from the delinked process segment, the latter two by an RTI instruction.

Wait(s,k): Again, PC and PS are dumped on the stack, and SP is stored in the descriptor. The status field obtains the positive value k (denoting "waiting"), and the last field is used to insert the descriptor in the queue starting from s. Insertion has to occur at a place determined by the waiting rank k. After this operation, any ready process is searched in the ring. The ring guarantees a fair scheduling strategy, because each process gets its turn. In particular, the one entering the waiting state automatically becomes the last one to be resumed.

In principle one could replace the ring by a queue of ready processes. This solution is better, if the number of processes in a system is large (and most of them are waiting most of the time). But otherwise, the overhead in delinking and reinserting descriptors in queues is a significant disadvantage.

Awaited(s): This is a simple test for the queue s to be empty, i.e. a test for the address value nil.

3.4. Interface Modules

The definition of an interface module specifies that at any moment at most one process may be executing some procedure defined in the module. "Executing" here means "actively executing", for the definition allows an exception: while a process is actively executing some interface procedure, other processes may be waiting, i.e. "executing" a wait statement within the module. Hence, wait statements represent some singular points within the module that are exempted from the strict mutual exclusion principle.

The reasoning for this rule, according to Hoare [3], goes as follows: If several processes simultaneously access common variables, the rules - in particular verification rules - of ordinary programming no longer apply. If accesses to common variables are restricted to specific areas - critical section, monitors, interface modules - and if mutual exclusion of simultaneous entry to such sections is guaranteed, then we can again deal with the common rules and laws of uniprogramming. In general, an invariant condition is established for an interface module that holds on entry, exit, and for each wait statement. Let us denote this condition with I. If a communication need arises between two processes P1 and P2, then this is because a certain other, supplementary condition B has arisen in, say P1, that needs to be communicated to P2; a signal is sent from P1 to P2. Hence we can write

P1: I and B {send(s)} I

The signal s may reactivate another process that is waiting for the signal. We must insist that the receiving process P2 can rely on B to hold when it resumes, for this is essentially the message carried by the signal. Hence the verification condition is

P2: I {wait(s)} I and B

Since the wait and signal statements occur in an interface module, we insist that P2 must continue, while P1 is delayed until P2 either leaves the module or "drops into" another wait statement. Hoare insists that at this point P1 is immediately resumed, because it seems intuitively right that an interface module (being a "critical region") should be "occupied" as rarely as possible. (see Fig.3) (Note that the post-condition of the send statement does not include B. This is because we wish to allow P2 to invalidate B after it has resumed upon receiving the signal.)

In our implementation, we have not obeyed this advice. Instead, we have exempted send statements from the mutual exclusion rule like wait statements. This solution has several advantages, but its consequences have not been fully explored. We merely wish to point out, that no difference arises between our treatment of send statements and that of Hoare, if a send statement is the last statement of an interface procedure. And this is the case in most examples that are commonly cited.

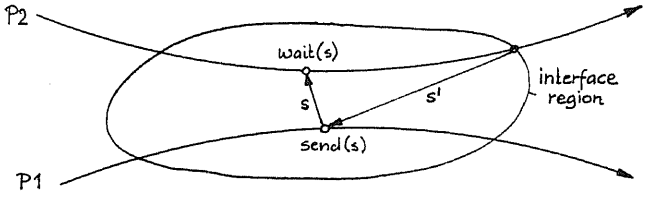

Fig.3 Processes with signal exchange

The advantage gained by exempting send statements from the mutual exclusion condition is that no mechanism for generating the signal s' (see Fig.3) is needed, and that no implicit processor switching occurs. A processor switches from one process to another only if executing an explicit wait or send statement. In fact, this is not only a definite implementation simplification, but also an easily comprehensible and helpful language rule.

The mechanism to assure mutual exclusion is now quite simple, and even trivial if Modula is implemented on a single-processor machine: no such mechanism is needed at all. If switches are restricted to wait and send statements, all processes except the one currently being served by the single processor are positioned at either a wait or a send statement. Since exactly these statements are exempted from mutual exclusion, no violation of the (relaxed) mutual exclusion rule can occur. (We have so far left out interrupts from our considerations on purpose.)

In principle, the interface module is therefore identical to a regular module (if implemented on a single-processor machine). It was originally introduced by Hoare (as a "monitor") with the intent that access to variables shared by several processes would be restricted to statements within the module. A compiler may easily check this rule. Again, we have not adopted it,

because - as shown by the Spacerace program - there are legitimate applications where such a language rule would be unnecessarily restrictive. In most cases it is not, and therefore the rule that accesses to shared variables be collected in an interface module is highly recommended as a programming principle (rather than a strictly enforced law).

4. Device Processes

An important goal in the development of Modula was to facilitate the effective programming of peripheral devices. Such devices usually operate concurrently and therefore not only require facilities for genuine multiprogramming but often also constitute the source of real-time programming problems. Processor and devices communicate via signals, e.g. starting pulses to initiate a device operation, and interrupts to signal the completion of a device operation. The interrupt now appears as an important hardware concept that must be dealt with, but cannot be directly expressed in a high-level language based on the notion of coherent sequential processes. Instead of trying to "make available" the interrupt in the undisguised form of some sort of jumps, actions executed by peripheral devices should be described by an explicit device statement (syntactically indistinguishable from "ordinary" statements). The compiler, knowing that such a statement is performed by a device, compiles code that initiates the device operation, sets up the interrupt return address, and releases the central processor. The address is such that an interrupt causes the process to resume with the statement following the device statement. Sandmayr [8] has adopted this strategy and provides two kinds of device statements, reflecting the simplicity of device handling available on his computer (HP 2115). We deviate from this scheme only slightly: The device statement does not include device initiation, which instead is expressed by separate, preceding commands. Hence, it suffices to introduce a single, parameterless device statement for all purposes. We shall call it "doio".

So far, processes could assume two possible states: ready and waiting (for a signal). Now a third state must be added: waiting for an interrupt (device signal). Such a simple approach should however be rejected, because the operation of processor switching due to an interrupt should be kept minimal, i.e. should not be augmented by additional "overhead" incurred merely because of an enforced mechanism of process administration. Interrupt handling in Modula should be precisely as efficient as if programmed in machine code. Efficiency can be improved by the following consideration: usually a whole sequence of device operations acknowledged by interrupts follow in short succession (e.g. reading all characters in a line). Hence, a process description may be left in the special third state until the entire sequence of interrupts has been received. This idea led to the device-control statement in [8], whose component statement S is executed without leaving the third state.

under dev control S end

S may contain loops, but is subjected to some obvious restrictions, such as exclusion of wait and send statements. The process state description then changes from "ready" to "under interrupt control" upon entry to the device control statement, and back upon exit. (The indicated device dev determines the interrupt location to be used within S).

The solution adopted in Modula goes even one step further in this direction: each statement S to be executed "under device control" must be declared as a process of its own. Of course, send and wait statements must then be admitted. A compiler must handle them in a different, special way that is designed to achieve the desired efficiency. Such processes are then called device processes. It follows that each such process is attached to a single device (whose associated interrupt location (or interrupt identification) is specified in its heading.) This concept reflects the customary practice of associating a single driver with each device. The driver or device process corresponds to the familiar concept of an interrupt routine. The driver is enclosed in an interface module (called a device module) together with all routines that communicate with the device and all details that are pertinent to the device. The module then serves to hide these details from the remainder of the program.

4.1. Process representation

The following details apply to the chosen strategy of a Modula implementation:

Device processes are identified by a descriptor similar to those of regular processes. However, they are not linked into the Ring. Their three states are described in Figs. 4 and 5.

This solution has the additional advantage that the number of registers to be saved upon an interrupt can be kept fairly small. The compiler may be advised, for example, to utilise only 2 or 3 work registers in device processes. Moreover, switching of regular processes may be performed without any work register saving and restoring whatsoever.

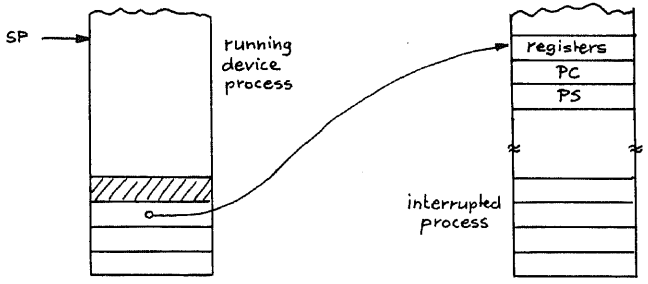

Fig.4. Device process in running state

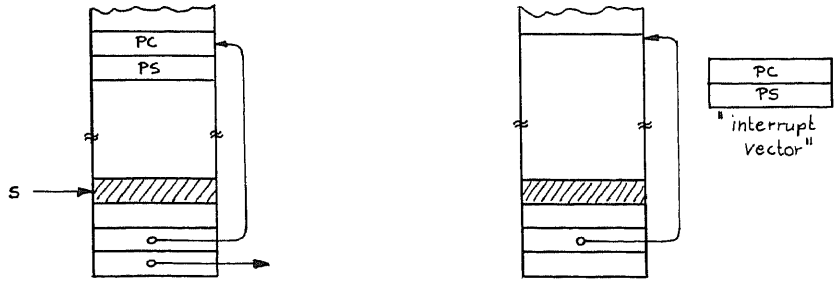

Fig.5. Device process in "waiting for signal" and "waiting for interrupt" states

4.2. Mutual exclusion, signalling, and process priorities

It is customary that during the execution of an interrupt routine further interrupts are withheld (delayed). An obvious approach to implement mutual exclusion in connection with device processes is therefore to simply shut off interrupts whenever a piece of program within a device module is executed. (Remember that a device module is automatically assumed to be an interface module.) Does this extremely simple and efficient solution suffice?

It does, provided the processor is returned to the interrupted process whenever the device process decides to release it. In principle, it must be released in three circumstances only: upon executing a doio, a wait, or a send statement. If the latter two are implemented accordingly - and they can be handled differently from waits and sends in regular processes - the solution is safe.

Many computers, including the PDP-11, feature an interrupt priority system. Each device is given (usually by the hardware) a fixed priority level. An interrupt from a device at level i then shuts out interrupts at levels $j \geq i$ (i.e. not necessarily all interrupts), until the program takes appropriate action. This notion is rather important in real-time programming, and must not be hidden by a high-level language.

The fact that a device has a certain interrupt priority i in effect means that its associated process has a priority i for obtaining the processor. Such a priority is therefore indicated in Modula in the heading of each device module. It signifies (in the PDP-11 implementation) that all program pieces within the device module are to be executed with that specified processor priority level. (This is another reason why device modules cannot be nested.)

The concept of a process priority raises another aspect about the rules of signalling. It was argued above that a process sending a signal must release the processor and pass it on to the receiving process. Should this also be true, if the sender

is a device process, i.e. one with a higher priority than the receiver? This would at least be counter-intuitive. We have therefore adopted the rule that a send statement executed by a driver does not release the processor, but merely marks the receiver as "ready". The above quoted axioms about wait and send statements therefore hold only provided the device process does not itself invalidate the condition signalled. We may formulate this restriction by the following general rule: a (device) process must never invalidate a condition signalled to a process of lower priority. This, however, does not appear as a handicap, if a driver operates in a simple producer/consumer constellation. And drivers usually satisfy this constraint. On the other hand, the advantage gained by this solution is significant, because the number of process switchings is thereby reduced, and the driver incurs no delay in reactivating its device. Perhaps the morale of this story is that in real-time multiprogramming the same verification axioms may be applied, provided certain additional constraints or rules of programming discipline are observed.

The different handling of regular and of device processes makes it necessary for a sender to be able to determine the category of the receiver. This requires a test of a descriptor bit. This test could be saved, if the compiler would be enabled to perform it. This would, however, require the introduction of a second kind of signals. We would then distinguish between signal variables upon which only regular processes may wait, and those on which device processes wait. Many programming applications have led to the assumption that such a restriction is in practice observed anyway. It even appears advisable to bind every device signal variable to one specific process.

5. The Nucleus

We call that part of a Modula program the Nucleus, which is identical for every program and resides permanently in the computer's main store. It is that part which embodies the mechanism for process administration including the routines for wait and send statements.

It must be the goal of any language (and of its implementation) for general systems programming to keep its nucleus very small. The structure and mechanism of the nucleus determines the overall concept of the language, and the bigger it is, the more does it restrict the flexibility of programs written in the language. A large nucleus is also an unmistakable symptom for built-in overhead. Nucleus routines should be minimal both in number and in size.

Modula has been designed with this advice in mind. Its nucleus consists of routines for the starting of processes, and for the wait and send statements. Moreover it contains some code for system initialization. The sizes of these routines are

routine	no. of instr.	no. of words
start	18	24
wait	27 (21)	3Ø
send	16 (9)	2Ø
initial	13	24

Hence the Nucleus has a total size of 98 words. (We do not consider the process descriptors as belonging to the Nucleus, although they are accessible to Nucleus routines only. Each descriptor occupies 4 words.) The lengths of the calling sequences of these routines are

routine	no. of instr.	no. of words
start	4	7
wait	3	5
send	4	5

The length of the calling sequence for the send routine contains a test for the signal value "not awaited". This avoids a call instruction in case the signal is not expected by any process. It appears essential that signalling is very cheap, if it has no effect.

The code representing the send, wait, and doio statements within device processes is expanded inline, and hence does not require any nucleus code. Its length is

macro	no. of instr.	no. of words
start	5+n	12+n
wait	8+n	16+n
send	5	6
doio	8+2n	14+2n

(n denotes the number of work registers available to device processes. They are saved and restored upon interrupt.)

Considering the brevity of this Nucleus, one is tempted to regard the Nucleus as a fix of the given hardware to accommodate the desired multiprogramming concept, rather than as a resident software support. It might very well be implemented by suitable microprograms (if that facility is available).

6. Miscellaneous Design Considerations

In this chapter, several issues will be discussed briefly that arose during the design of the language Modula and its first implementation. The list of these issues is neither exhaustive nor ordered according to their length of debate. (Neither did the length of discussions have a significant correlation to the importance of the issues.)

6.1. Statement structures

As far as the syntax is concerned, Modula deviates from its ancestor Pascal in an essential respect: statement structures consistently follow a single guiding rule. Every structured statement begins with a keyword (uniquely identifying the kind of structure) and ends with a closing symbol. The effect is shown for the while statement:

Pascal:	Modula:
`while B do S`	`while B do S end`
`while B do` `  begin S1; S2 end`	`while B do S1; S2 end`

The decisive consideration was that, if a statement is to be inserted in a program, no other symbols (such as a begin) would have to be inserted (deleted) in addition to the statement itself. The rule that every structured statement had to end with a closing symbol (usually end) had, however, the following consequence: Consider the frequent case of a cascaded conditional such as (written in Pascal or Algol 60)

```
if B1 then S1 else
   if B2 then S2 else
      ...
      if Bn then Sn else S
```

This now has to be expressed by the awkward construct

```
if B1 then S1 else
   if B2 then S2 else
      ...
      if Bn then Sn else S
      end
         ...
   end
end
```

It is quite natural in this case to introduce a construct that avoids the nesting of structures and eliminates the sequence of end symbols:

```
if B1 then S1
elsif B2 then S2
      ...
elsif Bn then Sn
else S
end
```

Note that this represents a genuine, clean solution in contrast to the fixup rule that "multiple ends" may be merged into a single end symbol.

6.2. Jumps and loop structures

The decision to omit jumps (goto statements) from Modula was a deliberate one; its aim is to explore the practicality of goto-free programming. It was taken without any claim to be the last word (and acknowledging the plain fact that many programmers will be happier and more at ease when a jump is available to them). It was evident, however, that at least one repetitve structure had to be included (in return), namely a loop statement allowing for one or several exits anywhere within the sequence of repeated component statements. The chosen form of such a loop statement appears as somewhat baroque. In its favor it can be stated that it allows the obvious representation of virtually all loop constructs (including exit actions specific to each exit point), and that its implementation is quite straight-forward. It has practically eliminated the need for labels and jumps. A very important use of the loop statement is the expression of a non-terminating repetition.

6.3. The data type bits

A conceptually appealing and efficient representation of bit-strings has been included in Pascal: the set structure. It is absent from Modula. This calls for an explanation. Reservations against adopting the set structure in Modula arose from the short wordlength in minicomputers, which is a natural limit for the cardinality of sets, if an efficient implementation is desired. A representation using multiple words for set variables was finally rejected, not so much on grounds of principle, but because of the desire to keep language size and compiler development effort in proportion to the available resources.

An analysis of the use of set structures in system programming indicated that access to individual bits is primarily needed for the following two purposes:

1. Operations on device registers, whose structure is defined by the hardware
2. Boolean arrays, such as storage reservation tables.

This led to the adoption of the single standard data type

```
bits = array 0:15 of Boolean
```

instead of a general set structure. Thereby the introduction of yet another concept is avoided. All operations applicable to Boolean arrays apply to bits. A compiler can easily recognise them as a case where dense data packing and use of particular instructions are applicable. As an extension, Modula admits the Boolean operators not only to components of bit variables, but to bit variables themselves, implying that they are executed (pairwise) on all components. This makes the direct use of the PDP-11 bit-instructions BIC, BIS, BIT and XOR possible.

The acceptance of a standard array type also calls for a de-notation of array constants. We have restricted such an array

constructor facility to the standard type bits, and instead of the obvious notation

[c0, c1, ... , c15]

where ci denotes the value of component i, we have retained the notation of sets, where

[x0, x1, ... , xn]

denotes the indices xi of all components having the value true. Moreover, this constructor is restricted: the xi must be constants. Therefore a compiler can always perform the test $0 \leq xi \leq 15$ for all elements, and it can directly construct an appropriate bitstring.

6.4. Integer division

The axiom of division

$$x/(-y) = (-x)/y = -(x/y)$$

specifies symmetry relative to 0 and is, to the author's knowledge, observed by all computers that feature a division instruction. It now happens that most divisions in system programming have a small constant as divisor; most frequent is a small power of 2. In these cases, a compiler should be able to employ a shift instruction for reasons of efficiency. But herewith arises a problem: if a computer uses two's complement representation for negative numbers, then division of negative dividends is not symmetric with respect to zero. There are the following possibilities for the language designer:

1. Test for the sign of the operand, i.e. compile code which includes the test(s) and corrects the quotient by adding or subtracting 1, if necessary.

2. In order to make the use of shift operations possible, eliminate the inconsistency by forbidding negative operands.

3. Introduce two distinct division operators in your language.

Solution 1 is unacceptable, because it more than offsets the efficiency gained by the use of a shift instruction. Solution 2 appears as uncunningly restrictive. Hence, there remains solution 3. In Modula, "regular" division is denoted by / and the division that can be implemented by shifts by div. Let x div y = q be the quotient and x mod y = r be the remainder. Then this division satisfies the following axiom for all integers x:

$$x = q*y + r \quad \text{and} \quad 0 \leq r < y$$

Note that the divisor must be strictly positive. Efficient compilation is now possible in those cases where it is most needed: division by small constants. Then the compiler can check $y > 0$, it can check whether the use of shift is possible, and otherwise use a division instruction with an appropriate correc-

tor. This is necessary for both the div and mod operations.

An important application of the mod instruction is the stepping of an index in a cyclic buffer with N elements. If the origin index is Ø, then

```
i := (i+1) mod N
i := (i-1) mod N
```

represent the stepping up and down of the index respectively. A single masking instruction can be used, if N is a power of 2.

6.5. A clock, a tick, or a pause?

Most modern computers feature a device that interrupts the processor in fixed time intervals, usually that of the line frequency. In the usual jargon, this device is called a clock. How should the clock be represented in a high-level language?

The most obvious way in Modula is to introduce a device module containing a "driver" process that periodically waits for the "clock interrupt" (denoted by "doio") and then sends a signal. For example:

```
loop doio; send(tick) end
```

One is very much tempted to declare such a process and the signal variable "tick" as standard components of Modula that are implicitly available. We have not done so, because frequently a clock process of a more complicated form, e.g.

```
loop doio; inc(time);
    while awaited(tick) do send(tick)
    end
end
```

might be preferred (or be necessary). By not declaring the tick signal to be a standard variable, the Modula programmer has the freedom (and the burden) to define his own clock process explicitly.

Remark concerning the PDP-11 implementation:
The restriction that signals must not be exchanged between device processes usually necessitates a second, regular clock process. The standardization of a minimal clock driver as a device process would therefore be even more justified. However, this consideration is particular to our PDP-11 implementation.

We have refrained from introducing a standard signal "tick", also because another solution might appear as more convenient for the programmer, namely the use of a standard procedure pause(n) that delays the calling process by n tick intervals. This procedure incarnates the genuine purpose of using a clock, namely the delaying of a process by a given time. The principal advantage of the pause procedure over the tick signal as a

predefined object is found in the reduction of the number of process switchings, if properly incorporated in the Nucleus.

When considering implementation techniques, an immediate temptation lies in introducing a third process state in addition to "ready" and "waiting" (we may call it "delayed"). The parameter n could be stored in a fifth field of the process record, and upon each clock interrupt, its value would be decremented for all delayed processes. A simpler solution lies in introducing a hidden signal variable (call it tick), and in interpreting each pause(n) as a wait(tick,n) statement. The clock interrupt routine then merely needs to decrement the delay ranks in all linked records, and to reactivate the first process, when its rank has reached Ø. (The wait statement performs the insertion at the appropriate place.)

7. Comments on the PDP-11

Every Modula program constitutes an abstract machine that (apart from some isolated parts called device modules) is defined entirely by the definition of the language Modula. Hence, what we call "language" is in effect an infinite collection of abstract machines or, more appropriately, a tool kit out of which arbitrary abstract machines can be engineered. We wish to make availabe a universal interpreter of these machines, and therefore build a compiler that enables an existing computer, in our case a PDP-11, to function as universal interpreter. The question then arises, to which degree the given computer is suitable for this purpose, or how natural the structure of the real machine is to mirror that of the abstract machine.

We have chosen the tool-kit Modula to reflect the dominant structural aspects of present computers quite truthfully, and to hide their peculiarities to a large degree, thereby creating a more systematic tool that is easier to comprehend and manage. Our subsequent considerations therefore focus on particular characteristics of the PDP-11 rather than its overall structure.

Some important points have already been mentioned by Bron [9], notably the well-designed subroutine call mechanism using a stack, and the old-fashioned concept of the condition codes. We give some further comments on these issues, and add a few items to the "negative list" in the hope that the criticism be understood as constructive and that it may help designers of hardware to recognize the issues.

The virtues of the subroutine call instruction JSR are that it performs what is needed (stacking the return address) and (almost) no more. JSR can be used to stack any register R and to place the return address into R. This is of some advantage when constant parameters are placed immediately following the jump instruction. This technique stems from practices of assembly coding and cannot be used in general. Our compiler does therefore not fully utilise the capabilities of the JSR instruction, since it always generates JSR PC,x. One detrimental effect in designing the JSR instruction more complicated than

needed, together with the desire for a matching return instruction, is the MARK instruction. Used in conjunction with RTS and added in later PDP-11 models as an afterthought, it is a beautiful example of poor "fixup engineering".

If a procedure p in a device module has to be called, then the current priority level of the processor must be saved before the jump is executed. The call is implemented by the 2 instructions

```
MOV PS,-(SP)      ;stack PS
JSR PC,p          ;stack PC and jump
```

It appears that a single instruction to obtain this effect is quite obviously missing, because the complementary instruction RTI (which unstacks PC and PS) is present. (The 4 (!) trap instructions appear identical to the proposed new instruction with the restriction that they jump to a fixed address.)

The concept of a condition code register serves to record certain frequently used predicates of every computed result. As such, we have no objections to it. Objectionable, however, is the lack of a suitable instruction that readily transfers these predicate values into data registers, preferably in the form of a Boolean value. Such a simple instruction would facilitate the compilation of Boolean expressions very considerably.

Apart from this, it appears that the PDP-11 structure is almost ideally suited for the efficient realization of the Modula concept of multiprogramming. This is mainly due to the ideas of a stack and SP-register and the way interrupts are implemented.

A few more remarks are in order concerning the operating of devices. The rule that each device has its own interrupt address is eminently sound and crucial. Also, the notion of priority levels and its realization are most appropriate. The desire to impose a systematic approach to device command and status representation is laudable. Nevertheless, in experimenting with various devices, I found it rare that knowledge in operating one device was immediately applicable to handling another. In particular, we found the following rules crucial, and postulate them as "axioms" for a sound concept of device handling:

1. Each device command (operation) is known to produce either no interrupt, or at least one interrupt.

2. Upon interruption, the interrogatable device status indicates whether or not another interrupt will follow.

3. Each interrupt is due to a distinct, preceding device command. (In other words: no interrupt arrives unexpectedly.)

4. Each device has one unique interrupt location. (A multiplexor handling several terminals, for example, may be considered as a single device.)

Unfortunately, the PDP-11 and/or its devices disregard these rules quite frequently and thereby often relegate the art of

device programming into the domain of heuristic experimentation. For example, we have a printer that requires a dc3 character to start its printchain drive. Sending a dc3 is acknowledged by an immediate interrupt (what for?), and a second interrupt follows when the chain has reached its regular speed, but only if the chain had not been running in the first place. There is no way to interrogate the chain drive status. Most output devices violate Rule 3: they interrupt the processor when the interrupt is initially enabled. Also, the card reader sends an unexpected interrupt when it is switched on. These cases have caused us to adopt the strict rule to switch off the interrupt enable bit after every interrupt (doio). This simply should not be necessary.

Because the PDP-11 attempts to systematise and unify device handling more than any other computer known to the author, deviations from the basic scheme are particularly visible and annoying. One such device is the GT-40 display unit. Its interrupts can trap to different locations (Rule 4), whether an interrupt is sent after device completion cannot be determined by the main processor, and interruptability cannot be enabled or disabled by setting or resetting a bit in a device register. The display unit's and the main processor's designs not only seem to originate from different engineers, but also from different eras of programming and different epoches of computer architecture.

The last point of criticism does not reflect so much on problems of computer design as on corporate attitude. I refer to the allocation of device registers in the regular storage addressing space. (The highest 4K addresses refer automatically to - mostly not installed - device interface registers instead of - often installed - memory.) As a result, if a customer decides to augment his PDP-11's store from 16K to 32K, he is sold 4K of store that he is prevented from using. (He is also told that buying 16K is cheaper than 12K!) Such an architectural decision betrays a company's low esteem of its customers: "they will buy even what they cannot use". The sad fact is of course that the customer's attitude justifies that decision: they do not dare to complain even if organised in big user unions. Is this a symptom of resignation, or incompetence, or simply indifference?

8. In Retrospect

I have tried to report on project Modula by not merely offering the final result (which may be not so final after all), but by describing the deliberations and considerations that led to it. In retrospect I recognise how ambitious this aim was. The design of a programming language and its compiler is a very large task in general. But if the subject is new, largely missing a solid scientific basis, and influenced by many rapid developments in technology, such a project is a risky foray with many surprises. Hence, its evolution cannot progress in a straight line. It is characterised by many short paths of exploration followed by retreats. In writing a report as this, one is not only tempted but forced to ignore many of these "incidents", although some of them still leave their traces. I should therefore warn the

reader that the development has not been as straight as it may appear from this account, and that many design decisions had been preceded by long arguments who helped to make us conscious of the motivations underlying the various alternatives.

The most important lesson perhaps is that conscious design decisions based on conscious motivations are very strongly influenced by the intended field of application. Being unable to consciously delimit this intended space of application is possibly the dominant reason for uncontrollable growth of language diversity and complexity [10]. If a language proves to be only marginally suitable for some application that was obviously not envisaged by its originator, we should muster the courage to build a new, truly adequate tool, instead of just grafting a fix onto the existing one.

References

1. N.Wirth, "Modula: A language for modular multiprogramming", Institut für Informatik ETH, Report No.18, March 1976.
2. N.Wirth, "The use of Modula", in this report.
3. C.A.R.Hoare, "Monitors: An operating system structuring concept", Comm.ACM 17, 10, 549-557 (Oct.1974).
4. P.Brinch Hansen, "The programming language Concurrent Pascal", IEEE Trans. Software Eng. 1, 2, 199-207 (June 1975).
5. P.Brinch Hansen, "The Solo operating system: a Concurrent Pascal program", and "Processes, monitors, and classes", Software - Practice and Experience 6, 141-149 and 165-200 (1976).
6. B.H.Liskov and S.Zilles, "Programming with abstract data types", ACM SIGPLAN Notices, 9, 50-59 (April 1974).
7. D.L.Parnas, J.E.Shore, D.M.Weiss, "Abstract types defined as classes of variables", ACM SIGPLAN Notices II, 2, 149-154 (1976) and Naval Research Lab. Report 7948, Wash. DC, April 1976.
8. H.Sandmayr, "Strukturen und Konzepte zur Multiprogrammierung, und ihre Anwendung auf ein System für Datenstationen (Hexapus), ETH-Diss. 5537, 1975.
9. C.Bron, W.deVries, "A Pascal compiler for PDP-11 minicomputers", Software - Practice and Experience 6, 109-116 (1976).
10. N.Wirth, "On the design of programming languages", Information Processing 74, (Proc. IFIP Congress 74), North-Holland Publ.

Berichte des Instituts für Informatik

Nr. 1	Niklaus Wirth:	The Programming Language Pascal (out of print)
Nr. 2	Niklaus Wirth:	Program development by step-wise refinement (out of print)
Nr. 3	Peter Läuchli:	Reduktion elektrischer Netzwerke und Gauss'sche Elimination
Nr. 4	Walter Gander, Andrea Mazzario:	Numerische Prozeduren I
Nr. 5	Niklaus Wirth:	The Programming Language Pascal (Revised Report) (out of print)
Nr. 6	C.A.R. Hoare, Niklaus Wirth:	An Axiomatic Definition of the Language Pascal (out of print)
Nr. 7	Andrea Mazzario, Luciano Molinari:	Numerische Prozeduren II
Nr. 8	E. Engeler, E. Wiedmer, E. Zachos:	Ein Einblick in die Theorie der Berechnungen
Nr. 9	Hans-Peter Frei:	Computer Aided Instruction: The Author Language and the System THALES (out of print)
r.10	K.V. Nori, U. Ammann, K.Jensen, H.H. Nägeli:	The PASCAL 'P' Compiler: Implementation Notes
r.11	G.I. Ugron, F.R. Lüthi:	Das Informations-System ELSBETH
Nr.12	Niklaus Wirth:	PASCAL-S: A Subset and its Implementation
Nr.13	U. Ammann:	Code Generation in a PASCAL Compiler
Nr.14	Karl Lieberherr:	Toward Feasible Solutions of NP-Complete Problems
Nr.15	E. Engeler:	Structural Relations between Programs and Problems
Nr.16	W. Bucher:	A contribution to solving large linear systems
Nr.17	Niklaus Wirth:	Programming languages: what to demand and how to assess them and Professor Cleverbyte's visit to heaven
Nr.18	Niklaus Wirth:	MODULA: A language for modular multiprogramming
Nr.19	Niklaus Wirth:	The use of MODULA and Design and Implementation of MODULA