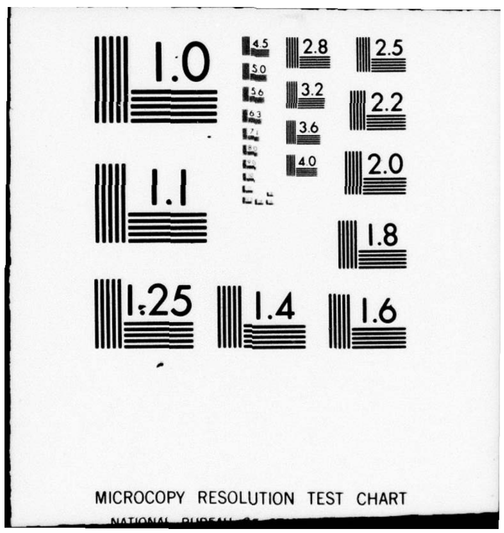
AD-A070 752 HONEYWELL INC MINNEAPOLIS MINN SYSTEMS AND RESEARCH --ETC F/G 9/2
SET OF SAMPLE PROBLEMS FOR DOD HIGH ORDER LANGUAGE PROGRAM. GRE--ETC(U)
APR 79 MDA903-77-C-0331

UNCLASSIFIED NL

1 OF 1
AD
A070 752

END
DATE
FILMED
8-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART

SET OF SAMPLE PROBLEMS FOR DOD HIGH ORDER LANGUAGE PROGRAM

GREEN SOLUTIONS

Honeywell, Inc.
Systems and Research Center
2600 Ridgway Parkway, Minneapolis, MN 55413

and

Cii Honeywell Bull
68 Route de Versailles
78430 Louveciennes, France

Contract No. MDA903-77-C-0331

April 1979

75p.

CONTENTS

1. Polled Asynchronous Interrupt

1.1 Sample Problem 1

Purpose:

An exercise to program a device and interrupt handler relying primarily upon polling techniques.

Problem:

(1) A channel handler will expect input by the function procedure call

'READ(DEVICE_NUMBER)'

and return a character from that device's input-stream.

(2) There should be a minimum delay from the time a character is introduced into the circular buffer and the time it may be accessible by a 'READ'. (the input will be displayed on the appropriate CRT by the reading process. Apparent simultaneity of hitting the key and appearance on the CRT is desired, i.e. the system should be reasonably efficient and thus provide good response-time.

(3) No input shall be lost.

Assumptions:

(1) A 16-bit, byte adressable machine

(2) At least 10 asynchronous input devices (keyboards) sharing I/O channel 0.

(3) A hardwired circular buffer of 128 bytes located at byte-location 500(8). Two pointers are provided in conjunction with the circular buffer:

HEADPOINTER - a pointer to the most recent input

TAILPOINTER - a pointer to the tail of the circular input queue

(4) the I/O channel will initialize both the HEAD- and the TAIL-pointer to the same location when the system is reset.

(5) A difference in the contents of the HEAD- and the TAIL-pointer indicates that input has occurred. Maintenance of the HEAD-pointer is the province of the I/O channel. Maintenance of the TAIL-pointer is the province of the channel handler.

(6) No interrupt shall occur when input is cleared except as noted in 7 below. The HEAD-pointer is incremented and the input stored in two bytes specified by the address contained in the HEAD-pointer.

(7) An interrupt will occur when the head pointer is pointing to the input-entry just below the entry indicated by the tail pointer to indicate that processing must occur to prevent loss of input.

(8) The interrupt location for channel Ø is 44Ø(8) and is two bytes in length to specify the location of the interrupt handling routine.

(9) An interrupt causes an implicit call of the specified routine. When processing of the interrupt has been completed, a return will cause the interrupted process to resume.

(1Ø) To simplify matters, assume
1) The context of the interrupted process is automatically saved and restored; that
2) no priority interrupt levels need be considered; and that
3) no clearing of the interrupt is required.

3.5(remark)

Each input consists of two bytes:
Byte Ø contains the ASCII character
Byte 1 contains the device identifier, Ø-9 to identify the sending keyboard

Guidelines:

It should be tried to formulate the program as hardware-independent as possible and clearly separate the interface to the hardware-dependent information.

1.2 Green Solution to Problem 1

(a) specific assumptions

HEAD and TAIL pointers must be located at specific addresses, since they are manipulated by hardware.

Since they are pointers, and the machine is byte-addressable, each operation must increment them by 2.

On occurrence of the interrupt, the response is to disable furtner input until at least one read has occurred. This is achieved by sending a disable signal on a separate line. Keyboards are reenabled by an enable signal.

(b) synopsis of the solution

The channel-handler is programmed as a task. Several similar channels could be provided by making this task a family.

The control of the input buffer and actual multiplexing is done by a nested task, POLLING: when some input has occurred, the first character to be output is transmitted to the appropriate program by an intermediate task family LOGICAL_KEYBOARD. Each member of this family performs a tight loop, successively receiving a character from POLLING, and sending it as a response to an external READ.

The user calls this READ indirectly by calling the value-returning procedure READ defined by the channel-handler, and which takes the device index as parameter.

```
task CHANNEL_HANDLER is
   type DEVICE is new INTEGER range 0 .. 9;
   procedure READ (D: DEVICE) return CHARACTER;
end;

task body CHANNEL_HANDLER is

   type ITEM is
      record
         CHAR: CHARACTER;
         DEV : DEVICE;
      end record;

   task LOGICAL_KEYBOARD(DEVICE'FIRST .. DEVICE'LAST) is
      entry READ   (C: out CHARACTER);
      entry DEPOSIT(C: in  CHARACTER);
   end;

   task POLLING;

   procedure READ(D: DEVICE) return CHARACTER is
      C: CHARACTER;
   begin
      LOGICAL_KEYBOARD(D).READ(C);
      return(C);
   end;

   task body LOGICAL_KEYBOARD is
      LAST_CHARACTER: CHARACTER;
      OK_TO_READ    : BOOLEAN := TRUE;
   begin
      loop
         select
            accept DEPOSIT(C: in CHARACTER) do
               if not OK_TO_READ then
                  LAST_CHARACTER := C;
               end if;
            end;
            OK_TO_READ := TRUE;

         or when OK_TO_READ =>
               accept READ(C: out CHARACTER) do
                  C := LAST_CHARACTER;
               end;
               OK_TO_READ := FALSE;
         end select;
      end loop;
   end;
```

```
   task body POLLING is

      subtype INDEX is INTEGER range 0 .. 63;
      BUFFER: array (INDEX'FIRST .. INDEX'LAST) of ITEM;
      HEAD, TAIL: INTEGER range 8#500 .. 8#677;
         -- initialized to 8#500 by I/O channel
      NEXT: INDEX;

      entry INTERRUPT;
      procedure ENABLE  is separate; -- restore keyboard input
      procedure DISABLE is separate; -- inhibits input from keyboard
      DISABLED: BOOLEAN := FALSE;
      POLLING_PERIOD: constant TIME := 0.1*SECONDS;

      for BUFFER use at 8#500;
      for ITEM use
         record at mod 2;
            CHAR at 0 range 0 .. 7;
            DEV  at 1 range 0 .. 7;
         end record;
      for HEAD use at  -- head address;
      for TAIL use at  -- tail address;
      for INTERRUPT use at 8#440;

   begin  -- POLLING
      loop
         select
            when HEAD = TAIL =>  -- buffer empty
               delay POLLING_PERIOD;
         or when not DISABLED =>
               accept INTERRUPT;
               DISABLED := TRUE;
               DISABLE;
               -- Other actions may be taken
         or when HEAD /= TAIL =>
               delay 0;  -- effect of a guarded else
               if TAIL=8#676 then TAIL := 8#500; else TAIL := TAIL+2; end if;
               NEXT := (TAIL - 8#500)/2;
               LOGICAL_KEYBOARD(BUFFER(NEXT).DEV).DEPOSIT(BUFFER(NEXT).CHAR);
               if DISABLED then
                  DISABLED := FALSE;
                  ENABLE;
               end if;
         end select;
      end loop;
   end POLLING;

begin
   initiate POLLING, LOGICAL_KEYBOARD(DEVICE'FIRST .. DEVICE'LAST);
end CHANNEL_HANDLER;
```

2. Priority Interrupt System

2.1 Sample Problem 2

<u>Purpose</u>:

An exercise to program an interrupt kernel supporting four levels of priority.

<u>Problem</u>:

An interrupt handling mechanism shall be described with the following functional capabilities:

(1) Higher priority interrupts should be able to preempt lower priority interrupt processes.

(2) As much processing as possible should be done with higher priority interrupts enabled. (Remark: In general, interrupts should only be disabled for the shortest possible time).

(3) A proper mechanism for the resumption of processing of preempted lower level interrupt (handler)s must be provided.

(4) To simplify matters, the body of each interrupt handler may be simulated, e.g. by a count of the interrupts for that priority level.

<u>Assumptions</u>:

(1) There are four interrupt priority levels: 0, 1, 2, 3.
The lower the number, the higher the priority.

(2) There is an interrupt vector located at 20(8) with 4 bytes for each priority level:
20(8): priority 0, 24(8): P1, 30(8): P2, 34(8): P3.

These locations specify the address of the interrupt handler for the corresponding priority level.

(3) The interrupt routine is invoked by an implicit call when the interrupt occurs.
At completion of the handler's processing, A return is to be performed.

(4) To simplify matters, assume that the interrupted processes' context is automatically saved and restored upon call and return. However, the information concerning the enablement and disablement of interrupts is not part of the context.

(5) Interrupts are enabled and disabled with a 'set interrupt instruction':

SIN<OPERAND>.

The interrupts to be enabled/disabled are specified by bits Ø-3 in the word addressed by the operand. The bit fields are:
Bit Ø(LSB) : priority Ø, bit 1 : priority 1, etc.

The values of these fields are:
Ø : disable 1 : enable

In order to disable all interrupts, perform an instruction "sin disable all", where the contents of DA = Ø.

(6) No clearing of the interrupts is required.

Guidelines:

Same as for Example 1. It should also be easy to replace the bodies of the interrupt-handlers. (e.g. at runtime, to allow for flexible reactions to an interrupt, according to circumstances).

2.2 Green Solution to Problem 2

The solution introduces a generic package INTERRUPT_HANDLING, to which the service routines can be passed as parameters. Thus, a typical use, for instance for counting interrupts, would be

```
declare
   COUNT_0, COUNT_1, COUNT_2, COUNT_3: INTEGER := 0;
   procedure INC_0 is
   begin
      COUNT_0 := COUNT_0 + 1;
   end;
   -- and similarly for INC_1, INC_2, INC_3
   package COUNT_INTERRUPTS is
      new INTERRUPT_HANDLING(INC_0, INC_1, INC_2, INC_3);
begin
   ...
end;
```

This package contains a family of four tasks, each one to execute the routine when an interrupt occurs. In addition, the task DISPATCH receives the interrupts, and releases the various HANDLER tasks when appropriate. Each handler can wait by calling the serve entry corresponding to its interrupt level (serve is a family of entries). The interrupts themselves are linked to the entries INT_i of DISPATCH.

THe physical masking and unmasking of interrupts is done by the Green runtime.

When an interrupt is received at a given level, it is recorded by setting the corresponding PENDING flag. When an interrupt is thus pending, and other interrupts are not pending at higher levels, the corresponding handler is released by accepting a call to the appropriate serve entry. When the routine has been executed by the handler, the interrupt is cleared by calling the RTI entry of DISPATCH, which will reset the PENDING flag.

Note that each handler runs at a different priority. This is necessary only to ensure that a higher priority interrupt can preempt execution of a handler for a lower priority one. The task priorities are not used to control when an interrupt can be accepted.

```
generic (procedure ROUTØ;
         procedure ROUT1;
         procedure ROUT2;
         procedure ROUT3)
package INTERRUPT_HANDLING;

package body INTERRUPT_HANDLING is

   type LEVEL is new INTEGER range Ø .. 3;

   task DISPATCH is

      entry INT_Ø;
      entry INT_1;
      entry INT_2;
      entry INT_3;

      entry SERVE(LEVEL'FIRST .. LEVEL'LAST);
      entry RTI  (LVL: LEVEL);

      for INT_Ø use at 8#2Ø;
      for INT_1 use at 8#24;
      for INT_2 use at 8#3Ø;
      for INT_3 use at 8#34;
   end;

   task HANDLER(LEVEL'FIRST .. LEVEL'LAST);

   task body HANDLER is
      ME: constant LEVEL := HANDLER'INDEX;
   begin
      SET_PRIORITY(SYSTEM'MAX_PRIORITY - (INTEGER(ME)+1));
      loop
         DISPATCH.SERVE(ME);
         case ME of
            when Ø => ROUTØ;
            when 1 => ROUT1;
            when 2 => ROUT2;
            when 3 => ROUT3;
         end case;
         DISPATCH.RTI(ME);
      end loop;
   end HANDLER;
```

```
    task body DISPATCH is

       PENDING: array(LEVEL'FIRST .. LEVEL'LAST) of BOOLEAN :=
                               (FALSE, FALSE, FALSE, FALSE);

    begin  -- DISPATCH
       SET_PRIORITY(SYSTEM'MAX_PRIORITY);
       loop
          select
             when not PENDING(0) =>
                accept INT_0;
                PENDING(0) := TRUE;

          or when not PENDING(1) =>
                accept INT_1;
                PENDING(1) := TRUE;

          or when not PENDING(2) =>
                accept INT_2;
                PENDING(2) := TRUE;

          or when not PENDING(3) =>
                accept INT_3;
                PENDING(3) := TRUE;

          or when PENDING(0) =>
                accept SERVE(0);

          or when PENDING(0 .. 1) = (FALSE, TRUE) =>
                accept SERVE(1);

          or when PENDING(0 .. 2) = (FALSE, FALSE, TRUE) =>
                accept SERVE(2);

          or when PENDING(0 .. 3) = (FALSE, FALSE, FALSE, TRUE) =>
                accept SERVE(3);

          or
             accept RTI(LVL: LEVEL) do
                PENDING(LVL) := FALSE;
             end;
          end select;
       end loop;
    end DISPATCH;

 begin -- initialization of INTERRUPT_HANDLING
    initiate DISPATCH, HANDLER(LEVEL'FIRST .. LEVEL'LAST);
 end INTERRUPT_HANDLING;
```

3. A Small File Handling Package

3.1 Sample Problem 3

Purpose:

An exercise to show how higher-level I/Ø functions can be constructed and used.

Problem:

Program a file system according to the following specifications:

(1) Files are built by producers who can perform the following operations:

```
CREATE   (FILENAME, ESTIMATED-SIZE)
WRITE    (FILENAME, DATA-AREA)
ENDWRITE (FILENAME)
```

The data contained in 'data-area' are written on the file with 'filename'. 'Data-area' can be anything from a single variable to an array of structures in memory.

Files are sequential, so each write adds a record to the end.
ENDWRITE signals completion of writing.

(2) Files are read by one or more consumers who use the following operation:

```
READ ( FILENAME, RECORD-NO., DATA-AREA )
```

Here, data are read from a given record from file 'filename'.

(3) Once all reading is complete, the file may be destroyed by calling:

```
DESTROY ( FILENAME )
```

Exceptions shall be raised in at least the following cases:

(A) If a producer wants to create a file with an already existing filename

(B) If a user wants to write on a nonexistent file

(C) If a consumer wants to read from a nonexistent file or from an existing file with a nonexistent record number

(D) If a file shall be destroyed while it is still used by somebody else.

Assumptions:

Assume a disk as storage medium.

Guidelines:

The design should prevent deadlock of file storage, allow disk operations to be scheduled according to any schedule (where the scheduler goes should be indicated), and prevent users from accessing anything but the above five operations.

3.2 Green Solution to Problem 3

The solution to the File Handling Package is developed in three levels : a high level introduces the notion of files of elements of a given type, and as such is generic. It defines the desired operations CREATE, DESTROY, READ, WRITE, END_WRITE. READ and WRITE operate on single elements of the file component type. At this level, file names are merely designated by a string of characters.

The second level defines an untyped file handling package : read and write operations work on arbitrary number of bytes. Files are designated by a character string and an unforgeable key, the use of which will be explained later.

The third level defines individual files as tasks, a file directory to map file names into task indices, and lastly a disk handler, which maps file storage onto disk storage.

Use of Keys

Since the FILE_IO package is generic, several instances of it can be created. All instances will be mapped on the FILE_MANAGER task. Since files are designated at the user level by a character string, the system must be capable of distinguishing between two files created by two different instances of FILE_IO, and, more importantly, to guarantee that a file created by one instance of FILE_IO is not written onto (or read) by another instance. For this purpose, a unique key is assigned to each instance of FILE_IO, and used as an additional parameter to identify files. Note that a different approach could be taken to ensure the type integrity of files, e.g. to use internal file names, allocated by CREATE. This is actually what is done in the user-level input-output facilities defined in the language.

Files, and the File Directory

In order to get as much idependence in the use of files as possible, each file is defined as an independent task : a task family FILE is defined in FILE_MANAGER, and a CREATE operation will associate a particular member of the family with the file name, and initiate that member. As a result, the correspondence between file names and file indices must be maintained. This is the domain of the task DIRECTORY.

Disk Handling

At the user level, read and write operations are performed on single elements of the component type of the file. At lower levels, however, these operations are defined in terms of the operand address in memory, and its size as a number of bytes. Furthermore, each file is divided into a certain number of blocks, each block being stored in one sector of the disk. In order to map file blocks onto disk blocks (i.e., to determine which disk sector corresponds to block b of file F), a disk map is maintained by the disk handler. Thus a disk request is made concerning a block of a file, and is translated to the appropriate disk address.

```
restricted(FILE_MANAGER)
generic (type ELEM)
package FILE_IO is

   subtype FILE_NAME is STRING;

   procedure CREATE    (F: FILE_NAME; SIZE: INTEGER);
   procedure DESTROY   (F: FILE_NAME);
   procedure WRITE     (F: FILE_NAME; DATA: ELEM);
   procedure END_WRITE(F: FILE_NAME);
   procedure READ      (F: FILE_NAME; RECORD_NO: INTEGER; DATA: out ELEM);

   INVALID_FILE       : exception renames FILE_MANAGER.INVALID_FILE;
   FILE_EXISTS        : exception renames FILE_MANAGER.FILE_EXISTS;
   DIRECTORY_FULL     : exception renames FILE_MANAGER.DIRECTORY_FULL;
   READ_ERROR         : exception renames FILE_MANAGER.READ_ERROR;
   WRITE_ERROR        : exception renames FILE_MANAGER.WRITE_ERROR;
   FILE_SIZE_EXCEEDED: exception renames FILE_MANAGER.FILE_SIZE_EXCEEDED;
   ILLEGAL_READ       : exception renames FILE_MANAGER.ILLEGAL_READ;
   ILLEGAL_WRITE      : exception renames FILE_MANAGER.ILLEGAL_WRITE;
   ILLEGAL_CLOSE      : exception renames FILE_MANAGER.ILLEGAL_CLOSE;

end FILE_IO;

package body FILE_IO is
-- All the calls of the typed operations of FILE_IO are converted
-- to the lower-level, untyped, operations of FILE_MANAGER.

   KEY: FILE_MANAGER.KEY_TYPE;

   procedure CREATE (F: FILE_NAME; SIZE: INTEGER) is
   begin
      FILE_MANAGER.CREATE(F, KEY, SIZE*ELEM'SIZE);
   end;

   procedure DESTROY(F: FILE_NAME) is
   begin
      FILE_MANAGER.DESTROY(F, KEY);
   end;

   procedure WRITE(F: FILE_NAME; DATA: ELEM) is
   begin
      FILE_MANAGER.WRITE(F, KEY, DATA'ADDRESS, ELEM'SIZE);
   end;

   procedure END_WRITE(F: FILE_NAME) is
   begin
      FILE_MANAGER.END_WRITE(F, KEY);
   end;

   procedure READ(F: FILE_NAME; RECORD_NO: INTEGER; DATA: out ELEM) is
      SIZE: constant INTEGER := RECORD_NO*ELEM'SIZE;
   begin
      FILE_MANAGER.READ(F, KEY, SIZE, DATA'ADDRESS, ELEM'SIZE);
   end;
```

```
begin
   FILE_MANAGER.GET_NEW_KEY(KEY);
end FILE_IO;

task FILE_MANAGER is

   subtype FILE_NAME is STRING;
   restricted type KEY_TYPE is private;

   entry CREATE (F: FILE_NAME; K: KEY_TYPE; SIZE: INTEGER);
   entry DESTROY(F: FILE_NAME; K: KEY_TYPE);

   procedure WRITE(F: FILE_NAME;
                   K: KEY_TYPE;
                   SOURCE_ADDR: INTEGER;
                   N_BYTES: INTEGER);

   procedure READ (F: FILE_NAME;
                   K: KEY_TYPE;
                   RECORD_ADDR: INTEGER;
                   DEST_ADDR: INTEGER;
                   N_BYTES: INTEGER);

   procedure END_WRITE(F: FILE_NAME; K: KEY_TYPE);

   entry GET_NEW_KEY(KEY: out KEY_TYPE);

   INVALID_FILE, FILE_EXISTS,    DIRECTORY_FULL,
   READ_ERROR,   WRITE_ERROR,    FILE_SIZE_EXCEEDED,
   ILLEGAL_READ, ILLEGAL_WRITE, ILLEGAL_CLOSE: exception;
private
   type KEY_TYPE is new INTEGER;
end FILE_MANAGER;

task body FILE_MANAGER is

   MAX_FILES: constant INTEGER := 100;
   type FILE_INDEX is new INTEGER range 1 .. MAX_FILES;

   LAST_KEY: KEY_TYPE := KEY_TYPE'FIRST;

   ------------------------------------------------------------
   -- The package DIRECTORY provides a mapping from extended
   -- file names (augmented with keys) to internal indices
   ------------------------------------------------------------
   package DIRECTORY is
      function  SEARCH(F: FILE_NAME; K: KEY_TYPE) return FILE_INDEX;
      procedure ADD   (F: FILE_NAME; K: KEY_TYPE) return FILE_INDEX;
      procedure REMOVE(F: FILE_NAME; K: KEY_TYPE);
   end DIRECTORY;
```

```
--------------------------------------------------------------
-- Each member of the family FILE separately controls
-- access to a given file. It is initiated when
-- the file is created, and terminates when the file
-- is destroyed. It can then be reused for another file.
--------------------------------------------------------------
task FILE(FILE_INDEX'FIRST .. FILE_INDEX'LAST) is
   procedure READ(RECORD_ADDR: INTEGER;
                  DEST_ADDR  : INTEGER;
                  N_BYTES    : INTEGER);

   entry WRITE(SOURCE_ADDR: INTEGER; N_BYTES: INTEGER);
   entry END_WRITE;
   entry OPEN(SIZE: INTEGER);
   entry CLOSE;
end FILE;


--------------------------------------------------------------
-- The DISK_HANDLER both provides a mapping between file
-- blocks and disk blocks, and gives access to the actual
-- disk operations.
--------------------------------------------------------------
task DISK_HANDLER is

   BLOCK_SIZE: constant INTEGER := 1024;
   MAX_BLOCKS: constant INTEGER := 800;

   subtype BLOCK_INDEX is INTEGER range 1 .. MAX_BLOCKS;
   subtype BYTE_OFFSET is INTEGER range 0 .. BLOCK_SIZE - 1;

   entry READ( DATA_ADDR: INTEGER;
               B: BLOCK_INDEX;
               DISPL: BYTE_OFFSET;
               N_BYTES: INTEGER);

   entry WRITE(DATA_ADDR: INTEGER;
               B: BLOCK_INDEX;
               DISPL: BYTE_OFFSET;
               N_BYTES: INTEGER);

   entry RESERVE(F: FILE_INDEX; N_BLOCKS: INTEGER);
   entry RELEASE(F: FILE_INDEX);

   function BLOCK_ADDR(F: FILE_INDEX;
                       BLOCK_NUM: INTEGER) return BLOCK_INDEX;

   INEXISTENT_BLOCK, DISK_FULL, DISK_ERROR: exception;

end DISK_HANDLER;
```

```
package body DIRECTORY    is separate;
task    body FILE         is separate;
task    body DISK_HANDLER is separate;

procedure WRITE(F: FILE_NAME;
                K: KEY_TYPE;
                SOURCE_ADDR: INTEGER;
                N_BYTES: INTEGER) is
   use DIRECTORY;
   FX: FILE_INDEX := SEARCH(F, K);
begin
   FILE(FX).WRITE(SOURCE_ADDR, N_BYTES);
exception
   when TASKING_ERROR =>
      raise INVALID_FILE;
end;

procedure END_WRITE(F: FILE_NAME; K:KEY_TYPE) is
   use DIRECTORY;
   FX: FILE_INDEX := SEARCH(F, K);
begin
   FILE(FX).END_WRITE;
end;

procedure READ(F: FILE_NAME;
               K: KEY_TYPE;
               RECORD_ADDR: INTEGER;
               DEST_ADDR: INTEGER;
               N_BYTES: INTEGER) is
   use DIRECTORY;
   FX: FILE_INDEX := SEARCH(F, K);
begin
   FILE(FX).READ(RECORD_ADDR, DEST_ADDR, N_BYTES);
end;
```

```
begin -- body of FILE_MANAGER
   loop
      begin
         select
            accept GET_NEW_KEY(KEY: out KEY_TYPE) do
               KEY := LAST_KEY;
            end;
            if LAST_KEY = KEY_TYPE'LAST then
               LAST_KEY := KEY_TYPE'FIRST;
            else
               LAST_KEY := LAST_KEY + 1;
            end if;
         or
            accept CREATE(F: FILE_NAME;
                          K: KEY_TYPE;
                          SIZE: INTEGER) do
               declare
                  FX: FILE_INDEX;
               begin
                  begin
                     FX := DIRECTORY.SEARCH(F, K);
                     raise FILE_EXISTS;
                  exception
                     when INVALID_FILE => null;
                  end;
                  FX := DIRECTORY.ADD(F, K);
                  initiate FILE(FX);
                  FILE(FX).OPEN(SIZE);
               end;
            end CREATE;
         or
            accept DESTROY(F: FILE_NAME; K: KEY_TYPE) do
               declare
                  FX: FILE_INDEX := DIRECTORY.SEARCH(F, K);
               begin
                  FILE(FX).CLOSE;
                  DIRECTORY.REMOVE(F, K);
               end;
            end DESTROY;
         end select;
      exception
         when others => null;
      end;
   end loop;
end FILE_MANAGER;
```

```
-------------------------------------------------------------
--  separately compiled bodies
-------------------------------------------------------------
restricted(FILE_MANAGER)
separate package body DIRECTORY is

   type NAME_REF  is access FILE_NAME;
   type DIR_ENTRY is
      record
         NAME: NAME_REF;
         KEY : KEY_TYPE;
      end record;

   FILE_MAP: array(FILE_INDEX'FIRST .. FILE_INDEX'LAST)
                  of DIR_ENTRY;

   function SEARCH(F: FILE_NAME; K: KEY_TYPE) return FILE_INDEX is
   begin
      for I in FILE_INDEX'FIRST .. FILE_INDEX'LAST loop
         if FILE_MAP(I).NAME /= null
            and then FILE_MAP(I).NAME.all = F
            and then FILE_MAP(I).KEY = K then
            return I;
         end if;
      end loop;
      raise INVALID_FILE;
   end SEARCH;

   procedure ADD(F: FILE_NAME; K: KEY_TYPE) return FILE_INDEX is
      FX: FILE_INDEX;
   begin
      begin
         FX := SEARCH(F, K);
         raise FILE_EXISTS;
      exception
         when INVALID_FILE => null;
      end;
      for I in FILE_INDEX'FIRST .. FILE_INDEX'LAST loop
         if FILE_MAP(I).NAME = null then
            FILE_MAP(I).NAME := new NAME_REF(F);
            FILE_MAP(I).KEY := K;
            return I;
         end if;
      end loop;
      raise DIRECTORY_FULL;
   end ADD;

   procedure REMOVE(F: FILE_NAME; K: KEY_TYPE) is
      FX: FILE_INDEX := SEARCH(F, K);
   begin
      FILE_MAP(FX).NAME := null;
   end REMOVE;
end DIRECTORY;
```

```
restricted(FILE_MANAGER)
separate task body FILE is
   use DISK_HANDLER;
   ME: constant FILE_INDEX := FILE'INDEX;

   FILE_SIZE    : INTEGER;
   LAST_WRITTEN: INTEGER := 0;
   DONE_WRITING: BOOLEAN:= FALSE;
   N_READERS    : INTEGER:= 0;
   CLOSED       : BOOLEAN := FALSE;

   entry START_READ;
   entry STOP_READ;

   procedure READ(RECORD_ADDR: INTEGER;
                  DEST_ADDR  : INTEGER;
                  N_BYTES    : INTEGER) is
      ADDR, CURRENT_BLOCK, BYTES_LEFT, BYTES_IN_BLOCK, OFFSET: INTEGER;
   begin
      if (not DONE_WRITING)
         or RECORD_ADDR > FILE_SIZE then
         raise ILLEGAL_READ;
      end if;
      ADDR := DEST_ADDR;
      CURRENT_BLOCK := (RECORD_ADDR/BLOCK_SIZE) + 1;
      OFFSET := RECORD_ADDR mod BLOCK_SIZE;
      BYTES_LEFT := N_BYTES;
      START_READ;
      -- the following loop distributes the read operation
      -- over all the blocks that contain part of the
      -- desired information
      loop
         if BLOCK_SIZE - OFFSET > BYTES_LEFT then
            BYTES_IN_BLOCK := BYTES_LEFT;
            BYTES_LEFT := 0;
         else
            BYTES_IN_BLOCK := BLOCK_SIZE - OFFSET;
            BYTES_LEFT := BYTES_LEFT - BYTES_IN_BLOCK;
         end if;
         begin
            DISK_HANDLER.READ(ADDR,
                              BLOCK_ADDR(ME, CURRENT_BLOCK),
                              OFFSET,
                              BYTES_IN_BLOCK);
         exception
            when DISK_ERROR | INEXISTENT_BLOCK =>
               raise READ_ERROR;
         end;
         exit when BYTES_LEFT = 0;
         ADDR := ADDR + BYTES_IN_BLOCK;
         CURRENT_BLOCK := CURRENT_BLOCK + 1;
         OFFSET := 0;
      end loop;
      STOP_READ;
   end READ;
```

```
begin  -- FILE
   accept OPEN(SIZE: INTEGER) do
      begin
         if SIZE mod BLOCK_SIZE = 0 then
            RESERVE(ME, SIZE/BLOCK_SIZE);
         else
            RESERVE(ME, SIZE/BLOCK_SIZE + 1);
         end if;
         FILE_SIZE := SIZE;
      exception
         when DISK_FULL =>
            raise DIRECTORY_FULL;
      end;
   end OPEN;

   while not DONE_WRITING loop  -- writing phase
      declare
         ADDR, BYTES_IN_BLOCK, BYTES_LEFT,
         CURRENT_BLOCK, OFFSET: INTEGER;
      begin
         select
            accept WRITE(SOURCE_ADDR: INTEGER; N_BYTES: INTEGER) do
               if LAST_WRITTEN >= FILE_SIZE then
                  raise FILE_SIZE_EXCEEDED;
               end if;
               ADDR := SOURCE_ADDR;
               CURRENT_BLOCK := ((LAST_WRITTEN + 1)/BLOCK_SIZE) + 1;
               OFFSET := (LAST_WRITTEN + 1) mod BLOCK_SIZE;
               BYTES_LEFT := N_BYTES;
               -- loop similar to the one done for READ
               loop
                  if BLOCK_SIZE - OFFSET > BYTES_LEFT then
                     BYTES_IN_BLOCK := BYTES_LEFT;
                     BYTES_LEFT := 0;
                  else
                     BYTES_IN_BLOCK := BLOCK_SIZE - OFFSET;
                     BYTES_LEFT := BYTES_LEFT - BYTES_IN_BLOCK;
                  end if;
                  begin
                     DISK_HANDLER.WRITE(ADDR,
                                        BLOCK_ADDR(ME, CURRENT_BLOCK),
                                        OFFSET,
                                        BYTES_IN_BLOCK);
                  exception
                     when DISK_ERROR | INEXISTENT_BLOCK =>
                        raise WRITE_ERROR;
                  end;
                  exit when BYTES_LEFT = 0;
                  ADDR := ADDR + BYTES_IN_BLOCK;
                  CURRENT_BLOCK := CURRENT_BLOCK + 1;
                  OFFSET := 0;
               end loop;
               LAST_WRITTEN := LAST_WRITTEN + N_BYTES;
            end WRITE;
```

```
        or
           accept END_WRITE;
           DONE_WRITING := TRUE;
           FILE_SIZE := LAST_WRITTEN;
        or
           accept CLOSE do
              raise ILLEGAL_CLOSE;
           end CLOSE;
        end select;
     exception
        when others => null;
     end;
  end loop;

  while (not CLOSED) loop -- reading phase
     begin
        select
           accept START_READ;
           N_READERS := N_READERS + 1;
        or
           accept STOP_READ;
           N_READERS := N_READERS - 1;
        or
           accept CLOSE do
              if N_READERS > 0 then
                 raise ILLEGAL_CLOSE;
              else
                 RELEASE(ME);
              end if;
           end;
           CLOSED := TRUE;

        or
           accept WRITE do
              raise ILLEGAL_WRITE;
           end WRITE;
        or
           accept END_WRITE do
              raise ILLEGAL_WRITE;
           end END_WRITE;
        end select;
     exception
        when others => null;
     end;
  end loop;
end FILE;
```

```
restricted(FILE_MANAGER)
separate task body DISK_HANDLER is
   type BLOCK_DESC is
      record
         FREE: BOOLEAN := TRUE;
         FX: FILE_INDEX;
         BX: BLOCK_INDEX;
      end;

   DISK_MAP: array(BLOCK_INDEX'FIRST .. BLOCK_INDEX'LAST)
                     of BLOCK_DESC;
   FREE_BLOCKS: INTEGER := MAX_BLOCKS;

   function BLOCK_ADDR(F: FILE_INDEX;
                          BLOCK_NUM: INTEGER) return BLOCK_INDEX is
      -- returns the disk address of the BLOCK_NUMth
      -- block of file F
   begin
      for I in BLOCK_INDEX'FIRST .. BLOCK_INDEX'LAST loop
         if DISK_MAP(I) = (FALSE, F, BLOCK_NUM) then
            return I;
         end if;
      end loop;
      raise INEXISTENT_BLOCK;
   end BLOCK_ADDR;


begin
   loop
      declare
         B : BLOCK_INDEX;
      begin
         select
            accept RESERVE(F: FILE_INDEX;
                                N_BLOCKS: INTEGER) do
            -- find the requested number of free blocks
            -- in the disk map, and allocate them to file f.
               if N_BLOCKS > FREE_BLOCKS then
                  raise DISK_FULL;
               else
                  FREE_BLOCKS := FREE_BLOCKS - N_BLOCKS;
                  B := 1;
                  for I in 1 .. N_BLOCKS loop
                     while not DISK_MAP(B).FREE loop
                        B := B + 1;
                     end loop;
                     DISK_MAP(B) := (FALSE, F, I);
                  end loop;
               end if;
            end RESERVE;
```

```
         or
            accept RELEASE(F: FILE_INDEX) do
            -- free all the blocks of file F in the disk map
               for B in BLOCK_INDEX'FIRST .. BLOCK_INDEX'LAST) loop
                  if (not DISK_MAP(B).FREE)
                     and DISK_MAP(B).FX = F then
                     DISK_MAP(B).FREE := TRUE;
                     FREE_BLOCKS := FREE_BLOCKS + 1;
                  end if;
               end loop;
            end RELEASE;
         or
            -- the rest of DISK_HANDLER is
            -- concerned with scheduling disk
            -- IO requests and is not given.
         end select;
      exception
         when others => null;
      end;
   end loop;
end DISK_HANDLER;
```

4. Dynamic Pictures

4.1 Sample Problem 4

Purpose:

An exercise to show how a graphic display of a dynamic situation can be programmed.

Problem:

On a display screen, a rectangular pattern of e.g. 10 horizontal and 10 vertical lines shall be drawn. (One might also imagine that the background is a simplified map.)

Within this grid, two movable objects shall be shown. They shall be discriminated either by color or by shape.

The speed and direction of each object shall be controlled by an input-device, e.g. a joystick.

There shall be a reset-button, which allows to bring the objects into some predefined position and a start-button, which causes them to move. If the objects collide, they shall start to blink and, after some seconds, return to their homing-position. This shall be equivalent to a reset.

Assumptions:

The 'start' and the 'reset' button shall be connected to the interrupt-handling mechanism of the underlying system in a way that different interrupts occur when different buttons are pressed.

The controlling input devices shall be purely passive, i.e. the position of the stick (left, right, forward, reverse) and its deviation from 'position zero', controlling the speed of the objects, have to be read in explicitly by the program. The position of the input-device shall be accessible to the program via two 16-bit registers (two bytes), one for each coordinate. Each byte shall contain a six-bit integer number (right adjusted) which represents the deflection in this particular direction in the moment of read-in. There exist all kinds of 'reasonable combinations' of these values, e.g. 15_right-60_forward, 56_left-10_reverse. The construction of the hardware shall be such that 'unreasonable combinations' cannot occur, like 10_left-20_right.

Guidelines:

The hardware characteristics of the display-device were mainly left out to prevent the solutions from becoming too lengthy.

The algorithms shall be independent of the actual characteristics of the display device, e.g. it shall not matter whether the display device has a vector generator or whether it is just able to plot random points. Whether the objects can be created by a pattern generator, or whether they have to be put together from points and/or lines. The necessary hardware dependencies should nevertheless be clearly identified and as well localized as possible.

The program shall be written and structured in a way that it will work with the most primitive display-hardware, e.g. a random-point display, which has a precision of 10 bits for each coordinate, but that the routines necessary for simulating more complex display capabilities can be easily removed.

To simplify matters, it can be assumed that the lowest level of output-routines need not be included in the example, i.e. as far as the problem is concerned, the output shall be regarded as completed, as soon as the co-ordinates of points (lines, objects, etc.) have been deposited as integer numbers in the appropriate buffers.

It is left to the designer how he chooses to implement the graphic representation, e.g. by formatting procedures (similar to character formats) operating on built-in data types or by special-data structures. It is also left to him how he wants to implement the emergency reaction, e.g. by a software-interrupt or by exceptions.

4.2 Green Solution to Problem 4

The objects are controlled by a process DISPLAY. This process provides a 1024x1024 grid which we assume to be read by an independent process (not given here) which uses it to refresh the screen. The objects are drawn on the grid by a procedure DRAW_OBJECT, also not given.

We provide a controller for each JOYSTICK, which performs the reading when required.

DISPLAY repeatedly computes the position of the objects, and has them drawn, except when they are blinking and must therefore not be drawn. An object can blink for at most 5 seconds, during which it will be alternately on and off for periods of one third of a second (the grid is recomputed every thirtieth of a second). Objects start blinking when they collide, or when they hit the grid limits.

```
task DISPLAY is
   type OBJECT_ID    is (SQUARE, CIRCLE);
   subtype SPEED     is INTEGER range -63 .. 63;
   subtype POSITION is INTEGER range 0 .. 1023;
   type OBJECT_INFO is
      record
         X,Y   : POSITION;
         BLINK : BOOLEAN;
      end record;

   type OBJECT is array(SQUARE .. CIRCLE) of OBJECT_INFO;
   procedure DRAW_OBJECT(WHICH : OBJECT_ID);

   GRID                : array(0 .. 1023, 0 .. 1023) of BOOLEAN;
   BASIC_PERIOD        : constant TIME := 0.033*SECONDS;
   BLINKING_DURATION : constant TIME := 5*SECONDS;
   BLINKING_PERIOD     : constant INTEGER := 10;

   entry START;
   entry RESET;

   for START use at ...; -- interrupt for start
   for RESET use at ...; -- interrupt for reset
end;
```

```
task JOYSTICK(SQUARE .. CIRCLE) is
   entry READ(SX,SY : out SPEED);
end;

task body JOYSTICK is
   type JOYSTICK_INFO is
      record
         LEFT_SPEED,
         RIGHT_SPEED,
         FORWARD_SPEED,
         REVERSE_SPEED : INTEGER range 0..63;
      end record;

   DEVICE   : array(SQUARE .. CIRCLE) of INTEGER := ...;
      -- contains the device number of each joystick;

   REGISTER : array(SQUARE .. CIRCLE) of JOYSTICK_INFO;
      -- the device registers for each joystick


   ME       : constant OBJECT_ID := JOYSTICK'INDEX;
   BYTE     : constant INTEGER := 8;

   for REGISTER use at ...;  -- address of device registers
   for JOYSTICK_INFO use
      record at mod 4;
         LEFT_SPEED    at 0      range 2 .. 7;
         RIGHT_SPEED   at 1*BYTE range 2 .. 7;
         FORWARD_SPEED at 2*BYTE range 2 .. 7;
         REVERSE_SPEED at 3*BYTE range 2 .. 7;
      end record;

begin
   loop
      accept READ(SX, SY : out SPEED) do
         SEND_CONTROL(DEVICE(ME), REGISTER(ME)'ADDRESS);
         delay 0.001*SECONDS;
         if REGISTER(ME).LEFT_SPEED > 0 then
            SX := -REGISTER(ME).LEFT_SPEED;
         else
            SX :=  REGISTER(ME).RIGHT_SPEED;
         end if;

         if REGISTER(ME).REVERSE_SPEED > 0 then
            SY := -REGISTER(ME).REVERSE_SPEED;
         else
            SY :=  REGISTER(ME).FORWARD_SPEED;
         end if;
      end READ;
   end loop;
end JOYSTICK;
```

```
task body DISPLAY is
   SPEED_UNIT : constant INTEGER := ...;
   -- assume a constant period : SPEED_UNIT gives
   -- the distance covesquare during this period by
   -- an object moving at speed 1.
   OB           : OBJECT;
   BLINK_ON     : BOOLEAN;
   BLINK_TIME   : TIME;
   SWITCH_BLINK : INTEGER;
   SX, SY       : SPEED;

   procedure INIT is
   begin
      for I in SQUARE .. CIRCLE loop
         OB(I).X     := 0;
         OB(I).Y     := 0;
         OB(I).BLINK := FALSE;
      end loop;
      BLINK_TIME := 0.0;
      SWITCH_BLINK := 0;
      BLINK_ON     := FALSE;
   end;

   procedure START_BLINK is
   begin
      if BLINK_TIME = 0.0 then
         BLINK_TIME  := BLINKING_DURATION;
         SWITCH_BLINK:= BLINKING_PERIOD;
         BLINK_ON    := FALSE;
      end if;
   end;

   procedure DRAW_OBJECT(WHICH : OBJECT_ID) is separate;

begin  -- DISPLAY
   accept RESET;
   loop
      INIT;
      accept START;
      loop
         select
            delay BASIC_PERIOD;
            if BLINK_TIME > 0.0 then
               BLINK_TIME := BLINK_TIME - BASIC_PERIOD;
               SWITCH_BLINK := SWITCH_BLINK - 1;
               if BLINK_TIME = 0.0 then
                  exit;
               elsif SWITCH_BLINK = 0 then
                  BLINK_ON := not BLINK_ON;
                  SWITCH_BLINK := BLINKING_PERIOD;
               end if;
            end if;
```

```
            for I in SQUARE .. CIRCLE loop
               if not OB(I).BLINK then
                  JOYSTICK(I).READ(SX,SY);
                  if   (OB(I).X = 0    and SX < 0)
                     or(OB(I).X = 1023 and SX > 0)
                     or(OB(I).Y = 0    and SY < 0)
                     or(OB(I).Y = 1023 and SY > 0) then
                        OB(I).BLINK := TRUE;
                        START_BLINK;
                  else
                     OB(I).X := OB(I).X + SPEED_UNIT*SX;
                     OB(I).Y := OB(I).Y + SPEED_UNIT*SY;
                  end if;
               end if;
            end loop;

            if OB(CIRCLE).X = OB(SQUARE).X
               and OB(CIRCLE).Y = OB(SQUARE).Y then   -- collision
                  OB(CIRCLE).BLINK := TRUE;
                  OB(SQUARE).BLINK := TRUE;
                  START_BLINK;
            end if;

            for I in SQUARE .. CIRCLE loop
               if not OB(I).BLINK or BLINK_ON then
                  DRAW_OBJECT(I);
               end if;
            end loop;

         or
            accept RESET;
            exit;

         end select;
      end loop;
   end loop;
end DISPLAY;
```

5. A Database Protection Module

5.1 Sample Problem 5

<u>Purpose</u>:

An exercise to demonstrate how complex synchronization mechanisms can be constructed on user level.

<u>Problem</u>:

A DBMS shall contain a module which controls access to given data areas.

The user (or a running process) shall be able to indicate whether he requires exclusive access to a certain part of a data base ('data-set') or whether he is willing to share this resource with other users (e.g. for reading).

The respective operations shall look like the following:

```
EXCLUSIVE (DATA-SET-NAME, PREEMPTION-PARAMETER);

SHARED (DATA-SET-NAME, PREEMPTION-PARAMETER);
```

By the following operation, the user shall be able to indicate that he no longer wants to use the data-set:

```
FREE (DATA-SET-NAME);
```

It shall be possible to specify, either by an executable statement at any time or by a kind of declaration at scope entry or at compile-time:

(A) Whether an exclusive reservation has priority over a shared reservation

(B) How many users may share a resource (this number may e.g. be limited by the length of some waiting queues)

(C) Which users may execute which kind of access

(D) Whether preemption is possible and, if not, whether an exception shall be raised in case of an attempt to use the preemption parameter.

(E) Whether different users have different priorities, and, if so, which ones

(F) Whether the demanding process shall just wait for the availability of the desired resource or whether in this case an exception shall be raised to allow for evasive action.

Note that 'user' may in this example also always mean: 'running process'.

The module shall be coded in the complete form it would require to put it into a library.

Proper procedures for cleanups shall be provided in case of preemption.

Assumptions:

No specific assumptions as far as the hardware is concerned.

Guidelines:

It is the implementor's option whether he prefers to provide one very general module with all these capabilities, or whether he wants to use generic facilities to create modules with a proper subset of the functionalities dependent of the actual requirements at the point of instantiation.

5.2 Green Solution to Problem 5

We implement the desired protection as follows:
Access to each data-set is controlled by a separate task in a family. On the other hand, each user is, upon login, attributed a task, e.g., an interactive command interpreter, which defines the desired functions EXECUTE, FREE and SHARED for that user. The task index can be (and is) used as a user-identification, and can be passed to other tasks executing on behalf of that user, if desired.

The system has a constant table of access rights, ACCESS_TABLE, initialized in the program.

One simplifying assumption has been made:

We assume that if a task makes a request with preemption, but cannot be granted the desired access immediately (because access has already been given to higher priority tasks, since we assume that a task cannot preempt one with a higher priority), then, if the task can wait, it is suspended, but loses the preemption attribute. Without this assumption, two more entries would be needed at each priority level.

We give a brief summary of what happens when a user wishes exclusive access to a data set (the workings are similar for shared access).

(1) The user calls EXCLUSIVE, giving it the data-set name and the preemption parameter.

(2) This procedure merely calls the homonymic procedure in the task$ corresponding to the data-set, passing the user-id as argument, together with the preemption parameter.

(3) Accesses are then validated, by checking the access list of that user for the given data set. If the request is not rejected, it is queued on an entry TEST_EXCLUSIVE for the appropriate priority.

(4) TEST_EXCLUSIVE will check if the request can be granted (possibly preempting other users), and if so, will actually honor it. If the request cannot be granted, a status flag passed as out parameter is set to false.

(5) If TEST_EXCLUSIVE did not satisfy the request, and the task is willing to wait, it is queued on the entry WAIT_EXCLUSIVE for the appropriate priority.

The correct handling of priorities is performed by appropriately guarding the various accept statement in the body of DATA_SET.

Lastly, we should describe how preemption is communicated to the user task. Each user task has a vector of booleans, one for each data-set. The flag corresponding to the data-set that was preempted is set to true, and the FAILURE exception is raised in the user task. When this exception is received, the vector is scanned, to determine if FAILURE was caused by preemption, and if so, from which data-set. If it turns out that all flags were false, then the user can deduce that the exception corresponded to a real failure, instead of a mere preemption. A sample body for a user process (in the form of an interactive command interpreter) is given to show how exceptions can be handled, here resulting in messages to be output on user terminal.

```
package DATA_SET_PROTECTION_MODULE is

    MAX_DATA_SET : constant INTEGER := 50;
    MAX_USER     : constant INTEGER := 100;
    MAX_PRIO     : constant INTEGER := 64;

    type DS_NAME   is new INTEGER range 1..MAX_DATA_SET;
    type USER_NAME is new INTEGER range 1..MAX_USER;
    type PRIO      is new INTEGER range 1..MAX_PRIO;

    type RIGHTS is
        (EXCLUSIVE_OK, SHARE_OK, PREEMPT_OK, WAIT_OK, SIGNAL_PREEMPT);

    type RIGHTS_LIST is
          array (RIGHTS'FIRST .. RIGHTS'LAST)      of BOOLEAN;
    type USER_LIST is
          array(USER_NAME'FIRST .. USER_NAME'LAST) of BOOLEAN;
    type DS_LIST is
          array(DS_NAME'FIRST .. DS_NAME'LAST)     of BOOLEAN;

    type RIGHTS_TABLE is
          array(DS_NAME'FIRST .. DS_NAME'LAST,
                USER_NAME'FIRST .. USER_NAME'LAST) of RIGHTS_LIST;

    type REQUEST_CODE is ...;
    -- different kinds of access requests that can be made

    PERMISSION_DENIED, PREEMPTION_DENIED, UNAVAILABLE: exception;

    ACCESS_TABLE : RIGHTS_TABLE := -- global access matrix;
    PRIORITY     : array (USER_NAME'FIRST .. USER_NAME'LAST)
                            of PRIO;

    task USER_PROCESS(USER_NAME'FIRST .. USER_NAME'LAST) is

       procedure EXCLUSIVE (DS : DS_NAME; PREEMPT : BOOLEAN);
       procedure SHARED    (DS : DS_NAME; PREEMPT : BOOLEAN);
       procedure FREE      (DS : DS_NAME);
       procedure REQUEST   (DS : DS_NAME; WHAT : REQUEST_CODE);
       function  GET_RIGHTS(DS : DS_NAME) return RIGHTS_LIST;

       PREEMPTED_DS : DS_LIST;

    end USER_PROCESS;

end DATA_SET_PROTECTION_MODULE;
```

```
restricted(TEXT_IO)
package body DATA_SET_PROTECTION_MODULE is

   task DATA_SET(DS_NAME'FIRST .. DS_NAME'LAST) is
      procedure EXCLUSIVE (WHO : USER_NAME; PREEMPT : BOOLEAN);
      procedure SHARED    (WHO : USER_NAME; PREEMPT : BOOLEAN);
      procedure FREE      (WHO : USER_NAME);
      procedure REQUEST   (WHO : USER_NAME; WHAT : REQUEST_CODE);
      function  GET_RIGHTS(WHO : USER_NAME) return RIGHTS_LIST;
   end DATA_SET;

   task body USER_PROCESS is
      use TEXT_IO;

      ME     : constant USER_NAME := USER_PROCESS'INDEX;
      DS_SET : DS_LIST := (DS_NAME'FIRST .. DS_NAME'LAST => FALSE);
         -- set of currently reserved data sets

      procedure EXCLUSIVE(DS : DS_NAME; PREEMPT : BOOLEAN) is
      begin
         DATA_SET(DS).EXCLUSIVE(ME,PREEMPT);
      end;

      procedure SHARED(DS : DS_NAME; PREEMPT : BOOLEAN) is
      begin
         DATA_SET(DS).SHARED(ME, PREEMPT);
      end;

      procedure FREE(DS : DS_NAME) is
      begin
         DATA_SET(DS).FREE(ME);
      end;

      procedure REQUEST(DS : DS_NAME; WHAT : REQUEST_CODE) is
      begin
         DATA_SET(DS).REQUEST(ME, WHAT);
      end;

      function GET_RIGHTS(DS : DS_NAME) return RIGHTS_LIST is
      begin
         return DATA_SET(DS).GET_RIGHTS(ME);
      end;
```

```
begin  -- body of USER_PROCESS
   loop
      begin
         -- read command
         -- execute command
      exception
         when UNAVAILABLE =>
            PUT("data set is busy");
         when PREEMPTION_DENIED =>
            PUT("preemption refused");
         when PERMISSION_DENIED =>
            PUT("access violation");
            for X in DS_NAME'FIRST .. DS_NAME'LAST loop
               if DS_SET(X) then
                  FREE(X);
               end if;
            end loop;
            raise; -- access violation is a fatal error
         when FAILURE =>
            declare
               BAD_ERROR : BOOLEAN := TRUE;
            begin
               for X in DS_NAME'FIRST .. DS_NAME'LAST loop
                  if PREEMPTED_DS(X) then
                     BAD_ERROR := FALSE;
                     PUT("you have been preempted from data set : ");
                     PUT(X);
                     PREEMPTED_DS(X) := FALSE;
                     exit;
                  end if;
               end loop;
               if BAD_ERROR then
                  PUT("killed!");
                  for X in DS_NAME'FIRST .. DS_NAME'LAST loop
                     if DS_SET(X) then
                        FREE(X);
                     end if;
                  end loop;
                  raise;
               end if;
            end;
      end;
   end loop;
end USER_PROCESS;
```

```
task body DATA_SET is

    ME : constant DS_NAME := DATA_SET'INDEX;

    MAX_SHARING : constant INTEGER := -- some constant;
    -- note : MAX_SHARING cannot be modified while the
    -- data-set is still active, i.e., still being accessed.

    USERS    : USER_LIST;
    -- set of users currently accessing the data-set
    MODE     : (IN_SHARED, IN_EXCLUSIVE) := IN_SHARED;
    N_USERS : INTEGER := 0;
    PRIORITY_FOR_EXCLUSIVE : BOOLEAN := TRUE;
    REQUEST_COUNT : array(PRIO'FIRST .. PRIO'LAST) of INTEGER
                          := (PRIO'FIRST .. PRIO'LAST => 0);

    entry TEST_EXCLUSIVE(PRIO'FIRST .. PRIO'LAST)
          (CHECK : out BOOLEAN; WHO : USER_NAME; PREEMPT : BOOLEAN);

    entry TEST_SHARED(PRIO'FIRST .. PRIO'LAST)
          (CHECK : out BOOLEAN; WHO : USER_NAME; PREEMPT : BOOLEAN);

    entry WAIT_EXCLUSIVE(PRIO'FIRST .. PRIO'LAST)(WHO : USER_NAME);

    entry WAIT_SHARED(PRIO'FIRST .. PRIO'LAST)(WHO : USER_NAME);

    entry RELEASE(WHO : USER_NAME);
    entry RESERVE(P : PRIO);

    procedure ASK(CHECK : out BOOLEAN;
                  WHO : USER_NAME;
                  PREEMPT, EXCL : BOOLEAN);

    procedure FREE(WHO : USER_NAME) is
    begin
       if not USERS(WHO) then
          raise PERMISSION_DENIED;
       else
          RELEASE(WHO);
       end if;
    end FREE;

    procedure REQUEST(WHO : USER_NAME; WHAT : REQUEST_CODE) is
    begin
       if not USERS(WHO) then
          raise PERMISSION_DENIED;
       else
          -- perform the access : no synchronization is necessary,
          -- since it has been achieved by the reservation requests
       end if;
    end REQUEST;
```

```
procedure EXCLUSIVE(WHO : USER_NAME; PREEMPT : BOOLEAN) is
   MY_RIGHTS : constant RIGHTS_LIST := ACCESS_TABLE(ME, WHO);
   ACCEPTED  : BOOLEAN;
begin
   if not MY_RIGHTS(EXCLUSIVE_OK) then
      raise PERMISSION_DENIED;
   elsif PREEMPT and not MY_RIGHTS(PREEMPTION_OK) then
      if MY_RIGHTS(SIGNAL_PREEMPT) then
         raise PREEMPTION_DENIED;
      else
         PREEMPT := FALSE;
      end if;
   end if;
   RESERVE(PRIORITY(WHO));
   TEST_EXCLUSIVE(PRIORITY(WHO))(ACCEPTED,WHO,PREEMPT);
   if not ACCEPTED then
      if not MY_RIGHTS(WAIT_OK) then
         raise UNAVAILABLE;
      else
         RESERVE(PRIORITY(WHO));
         WAIT_EXCLUSIVE(PRIORITY(WHO))(WHO);
      end if;
   end if;
end EXCLUSIVE;

procedure SHARED (WHO : USER_NAME; PREEMPT : BOOLEAN) is
   MY_RIGHTS : constant RIGHTS_LIST := ACCESS_TABLE(ME, WHO);
   ACCEPTED  : BOOLEAN;
begin
   if not MY_RIGHTS(SHARE_OK) then
      raise PERMISSION_DENIED;
   elsif PREEMPT and not MY_RIGHTS(PREEMPTION_OK) then
      if MY_RIGHTS(SIGNAL_PREEMPT) then
         raise PREEMPTION_DENIED;
      else
         PREEMPT := FALSE;
      end if;
   end if;
   RESERVE(PRIORITY(WHO));
   TEST_SHARED(PRIORITY(WHO))(ACCEPTED,WHO,PREEMPT);
   if not ACCEPTED then
      if not MY_RIGHTS(WAIT_OK) then
         raise UNAVAILABLE;
      else
         RESERVE(PRIORITY(WHO));
         WAIT_SHARED(PRIORITY(WHO));
      end if;
   end if;
end SHARED;

function GET_RIGHTS(WHO : USER_NAME) return RIGHTS_LIST is
begin
   return ACCESS_TABLE(ME, WHO);
end GET_RIGHTS;
```

```
-----------------------------------------------------
-- The procedure ASK, which is not visible, performs
-- all the checks needed to permit a request.
-- If preemption must be done, it is performed here.
-----------------------------------------------------
procedure ASK(CHECK   : out BOOLEAN;
              WHO     : USER_NAME;
              PREEMPT : BOOLEAN;
              EXCL    : BOOLEAN) is
   TEMP_USERS : USER_LIST := USERS;
   CNT        : INTEGER := 0;

begin
   if MODE = IN_EXCLUSIVE or N_USERS = MAX_SHARING then
      -- preempt necessary if requested.
      if PREEMPT then
         for X in USER_NAME'FIRST .. USER_NAME'LAST loop
            if USERS(X) and PRIORITY(X) > PRIORITY(WHO) then
               CNT := CNT + 1;
               TEMP_USERS(X) := FALSE;
               if EXCL then
                  CHECK := FALSE;
                  return;
               end if;
            end if;
         end loop;
         if CNT = MAX_SHARING then
            CHECK := FALSE;
            return;
         elsif EXCL then  -- preempt all
            for X in USER_NAME'FIRST .. USER_NAME'LAST loop
               if USERS(X) then
                  USERS(X) := FALSE;
                  USER_PROCESS(X).PREEMPTED_DS(ME) := TRUE;
                  raise USER_PROCESS(X).FAILURE;
               end if;
            end loop;
            N_USERS := 0;
         else  -- preempt one of the possible users
            declare
               X : USER_NAME;
            begin
               X := CHOOSE(TEMP_USERS);
               -- assume CHOOSE returns an element
               -- chosen non_deterministically in the set.
               USERS(X) := FALSE;
               USER_PROCESS(X).PREEMPTED_DS(ME) := TRUE;
               raise USER_PROCESS(X).FAILURE;
               N_USERS  := N_USERS - 1;
            end;
         end if;
      else
         CHECK := FALSE;
         return;
```

```
            end if;
         end if;
         USERS(WHO)  := TRUE;
         N USERS     := N USERS + 1;
         CHECK       := TRUE;
         if EXCL then
            MODE := IN_EXCLUSIVE;
         end if;
      end ASK;

   begin  -- body of DATA_SET
      loop
         accept RESERVE(P : PRIO) do
            REQUEST_COUNT(P)  := REQUEST_COUNT(P) + 1;
         end;

         -- flush the RESERVE queue
         loop
            select
               accept RESERVE(P : PRIO) do
                  REQUEST_COUNT(P)  := REQUEST_COUNT(P) + 1;
               end;
            else
               exit;
            end select;
         end loop;

         for P in reverse PRIO'FIRST .. PRIO'LAST loop
            if REQUEST_COUNT(P) > 0 then
               select
                  accept TEST_EXCLUSIVE(P)(CHECK : out BOOLEAN;
                                           WHO : USER_NAME;
                                           PREEMPT : BOOLEAN) do
                     ASK(CHECK,WHO,PREEMPT,TRUE);
                  end;

               or when not PRIORITY_FOR_EXCLUSIVE
                        or TEST_EXCLUSIVE'COUNT = 0 =>
                     accept TEST_SHARED(P)(CHECK : out BOOLEAN;
                                           WHO     : USER_NAME;
                                           PREEMPT : BOOLEAN) do
                        ASK(CHECK,WHO,PREEMPT,FALSE);
                     end;

               or when N_USERS = 0 =>
                     accept WAIT_EXCLUSIVE(P)(WHO : USER_NAME) do
                        USERS(WHO)  := TRUE;
                     end;
                     N_USERS  := 1;
                     MODE     := IN_EXCLUSIVE;

               or when (MODE = IN_SHARED and N_USERS < MAX_SHARING)
                     or (not (N_USERS = 0 and PRIORITY_FOR_EXCLUSIVE
                           and(TEST_EXCLUSIVE(P)'COUNT>0
                              or WAIT_EXCLUSIVE(P)'COUNT>0))) =>
```

```
                    accept WAIT_SHARED(P)(WHO : USER_NAME) do
                       USERS(WHO) := TRUE;
                    end;
                    N_USERS := N_USERS + 1;
                 end select;

                 REQUEST_COUNT(P) := REQUEST_COUNT(P) - 1;
                 exit;  -- go and process newly arrived requests
              end if;
           end loop;
           loop
              select
                 accept RELEASE(WHO : USER_NAME) do
                    USERS(WHO) := FALSE;
                 end;
                 N_USERS := N_USERS - 1;
                 if MODE = IN_EXCLUSIVE then
                    MODE := IN_SHARED;
                 end if;
              else
                 exit;
              end select;
           end loop;
        end loop;
     end DATA_SET;
  end DATA_SET_PROTECTION_MODULE;
```

6. A Process Control Example

6.1 Sample Problem 6

<u>Purpose</u>:

An exercise to test interactions between parallel processing and exception handling.

<u>Problem</u>:

Assume four processes:

Process a which reads in data from the environment and stores them in a buffer area,

process b which processes the data it finds in the buffer area according to some algorithm and stores them in a 'result area', and

process c which produces output as a consequence of these data (either in human-oriented form or as control-output for the process to be controlled).

Process d monitors and controls these three (and possibly other) processes and interacts with the operator via a keyboard console.

It shall be further assumed that process a and process b interact in the following specific way:

The buffer is organized as a 'double-buffer', i.e., after one of its two areas has been filled by process a, process b is notified and starts to read out of the buffer. Process a continues by depositing data in the second buffer area. If this is full, process a tries to deposit data in the first area again. Process b, in turn, notifies process a after having read one data area.

It is illegal to read a buffer area which has not previously been filled and to write into a buffer area which has not been completely read (except in the initialization phase).

The program shall be structured in a way that it is possible to replace process a by appropriate hardware without having to change the program parts for processes b, c, and d.

It shall also be possible to terminate process a and b at any time without losing data, i.e. before termination a cleanup operation shall be invoked which causes processing of any remaining data in either of the two buffer areas.

Assumptions:

No particular assumptions as far as hardware is concerned.

The buffers and the 'result area' can be organized as arrays.

Guidelines:

To simplify matters, it can be assumed that actual input-output, i.e. the communication with the hardware, as well as the processing of the data in process b is done by given library routines.

The algorithm in process d may also be described in a highly summarized form, because this is not what the example is to test.

6.2 Green Solution to Problem 6

The solution presented here shows the flexibility and power of the Green tasking facilities. It is somewhat unconventional, in that each buffer is controlled by a separate task. The proper handling of multiple buffers is achieved exclusively by task synchronization.

The tasks A, B, C, and D are defined inside the package BUFFERS. The buffer size, maximum number of buffers, and element type are defined there and could be modified (i.e., the solution will work for an arbitrary number of buffers).

Task D repeatedly receives control commands, and can in particular receive an interrupt through the INTERRUPT entry.

Task C repeatedly performs the same action (e.g. display an image on a screen), using data found in RESULT_AREA. C is not synchronized with any other task, as it is acceptable that the contents of RESULT_AREA be changed while it is being read.

Task B repeatedly receives a buffer through its entry NEXT_BUFFER. The number of elements contained in the buffer is also passed as parameter. If the buffer is not full, it is the last one, and is an indication that B should terminate.

Each buffer is controlled by a task of the family WORKER. These tasks are local to task A, and are synchronized by A as follows. Each worker waits on an accept statement for the entry RELEASE. A call to RELEASE signals the worker to enter a writing phase. When the worker has filled its buffer, it calls the entry FULL of A, to indicate that A can release the next worker, and it then sends its buffer to B by calling the entry NEXT_BUFFER of B. If all buffers are full, A will wait on the call to RELEASE. If all buffers are empty, A will wait on the accept statement for FULL.

Termination is handled as follows: when an interrupt is received by D, the entry STOP of A is called. A will call in turn the entry STOP of the worker that is currently reading. The worker will stop reading and send the current contents of its buffer to B, while holding A in a rendezvous. When the incomplete buffer has been accepted by B, the worker is released, which will also release A. At this point, all input has been transmitted to B, and A can safely abort all workers.

```
package BUFFERS is
   MAX_BUFFERS: constant INTEGER := 2;
   MAX_SIZE   : constant INTEGER := 1000;

   subtype BUFF_RANGE is INTEGER range 1 .. MAX_SIZE;
   subtype BUFF_NAME  is INTEGER range 1 .. MAX_BUFFERS;
   type ITEM       is new CHARACTER;
   type BUFF_TYPE is array(1 .. MAX_SIZE) of ITEM;

   RESULT_AREA:  -- working area filled by B and read by C

   task A is
      entry STOP;
   end;
   task B is
      entry NEXT_BUFFER(BUF: BUFF_TYPE; N_ELTS: INTEGER);
   end;
   task C;
   task D;
end;
```

```
package body BUFFERS is
  procedure READ(C : in out ITEM) is separate;

  task body A is
     FILLING: BUFF_NAME := 1;
     entry FULL;

     task WORKER(BUFF_NAME'FIRST .. BUFF_NAME'LAST) is
        entry RELEASE;
        entry STOP;
     end;

     function NEXT(B: BUFF_NAME) return BUFF_NAME is
     begin
        return (B mod MAX_BUFFERS) + 1;
     end;

     task body WORKER is
        BUFFER: BUFF_TYPE;
        ME    : constant INTEGER := WORKER'INDEX;
     begin
     <<FILL_BUFFER>> loop
           accept RELEASE;
           for I in 1 .. MAX_SIZE loop
              select
                 accept STOP do
                    B.NEXT_BUFFER(BUFFER, I-1);
                 end STOP;
                 exit FILL_BUFFER;
              else
                 READ(BUFFER(I));
              end select;
           end loop;
           A.FULL;
           B.NEXT_BUFFER(BUFFER, MAX_SIZE);
        end loop FILL_BUFFER;
     end WORKER;

  begin -- A
     initiate WORKER(BUFF_NAME'FIRST .. BUFF_NAME'LAST);
     loop
        WORKER(FILLING).RELEASE;
        select
           accept FULL;
           FILLING := NEXT(FILLING);
        or when FULL'COUNT = Ø =>
           accept STOP;
           WORKER(FILLING).STOP;
           exit;
        end select;
     end loop;
     abort WORKER(BUFF_NAME'FIRST .. BUFF_NAME'LAST);
  end A;
```

```
task body B is
begin
   loop
      accept NEXT_BUFFER(BUF: BUFF_TYPE; N_ELTS: INTEGER) do
         -- compute values using BUF, and store
         -- results in RESULT_AREA
         exit when N_ELTS < MAX_SIZE;
      end NEXT_BUFFER;
   end loop;
end B;

task body C is
begin
   loop
      -- perform some actions, using contents of RESULT_AREA
   end loop;
end C;

task body D is
   entry INTERRUPT;
   for INTERRUPT use at ...;
begin
   loop
      select
         accept INTERRUPT;
         A.STOP;
      or -- accept other controls, or do other things
         ...
      end select;
   end loop;
end D;

begin -- initialization of BUFFERS
   initiate A, B, C, D;
end;
```

7. Adaptive Routing Algorithm for a Node within a Data Switching Network

7.1 Sample Problem 7

<u>Purpose</u>:

Test for language suitability for multicomputer and communications applications.

<u>Problem</u>:

Develop the program for a multiprocessor within one node of a data switching network to maintain the tables of

(1) distances,

(2) minimum delay time, and

(3) routing for the following adaptive routing algorithm:

Each node in a network maintains a table of distances and a table of minimum delay times between itself and all other nodes. The distance metric is the minimum number of hops required to reach each other node. Both tables are maintained through updates in the form of table exchanges which occur only between neighbor nodes (nodes of distance, one). Each node maintains a routing table which directs routing through that neighbor node which achieves the minimum delay time.

In parallel with, and at the same periodic rate as this computing process, separate computing processes at each node are computing the minimum delay times to neighbors, and reading into computer memory the updated distance table of each neighbor, and the updated minimum delay time table of each neighbor. Initially each node knows only the distance to each neighbor, which is one, and the minimum delay time to each neighbor. Other distances and minimum delay times are initially considered infinite. Each node iteratively builds up its own distance and minimum delay time tables from the distance and minimum delay time tables exchanged with its neighbors, and updates tables containing such information about itself. Other computing processes transmit this information between such neighbors. Hence, the routing table at each node is established and periodically updated adaptively from the minimum delay times.

When a link is broken or established, a separate computing process at each of the two former or new neighbors corrects the distance and minimum delay time tables.

The reason a distance table must be mined is that if the network is disconnected the algorithm causes the distance between disconnected nodes to increase without limit. Thus whenever the distance between two nodes becomes greater than the number of nodes in the network, this distance and minimum delay time is considered infinite, and the node is considered unreachable.

In the example program, consider that the number of nodes in the network, the neighbors of the programmed node, and the periodic update interval are constants known at compile time.

Assumptions.

None as far as the hardware is concerned.

Guidelines:

The actual interchange between the nodes can be assumed to be performed by given library routines.

7.2 Green Solution to Problem 7

At each node in the network, a task TABLE keeps a table of distances and minimum delay times between the node itself and all other nodes (NODE_TABLE). This task also maintains a routing table(NODE_ROUTE).

These tables are periodically updated by a task UPDATE_TABLE which reads all neighboring tables and computes new tables.

It is assumed that each node knows its neighbors through the table NEIGHBOR and knows the delay time between itself and these nodes through NEIGHBOR_DEL. Whenever a link is established, the delay time between the two nodes is passed to the updating task.

The table of a neighbor node is obtained through a call to the library routine READ_NEIGHBOR_TABLE which uses the entry READ of the corresponding TABLE task. When the new tables have been computed, the updating is done by copying them into the TABLE task of the current node through the entry WRITE. The entry GET_ROUTE of a task TABLE is used to interrogate the routing table. The entries ESTABLSH_LINK and BREAK_LINK of the updating task are used respectively when a new link is created or when a link is broken.

```
package NETWORK_INFO is

   subtype NODE is INTEGER range 1..50;
   subtype DISTANCE is INTEGER range
               NODE'FIRST - 1 .. NODE'LAST;
   subtype EXTENDED_NODE is DISTANCE;

   type NODE_INFO is
        record
           DEL  : INTEGER;
           DIST : DISTANCE;
        end record;

   type ALL_INFO   is array(NODE) of NODE_INFO;
   type ROUTE_INFO is array(NODE) of EXTENDED_NODE;

   INFINITY       : constant INTEGER := INTEGER'LAST;
   NULL_NODE      : constant EXTENDED_NODE := NODE'FIRST - 1;
   BASIC_PERIOD   : constant INTEGER := 60*SECONDS;

   -- additional information relative to the current node
end NETWORK_INFO;

restricted(NETWORK_INFO)
procedure MAIN is
   use NETWORK_INFO;

   ME : constant NODE := -- index of the node-id of this node;

   NEIGHBOR     : array(NODE'FIRST .. NODE'LAST) of BOOLEAN := ...;
   NEIGHBOR_DEL : array(NODE'FIRST .. NODE'LAST) of INTEGER := ...;
   --  minimum delay time to the neighbors

   package LIBRARY is

      procedure READ_NEIGHBOR_TABLE(N : NODE;
                HIS_TABLE : out ALL_INFO);
      -- this procedure "tele"-issues a READ for
      -- the process TABLE of node N
   end LIBRARY;

   task UPDATE_TABLE is
      entry ESTABLISH_LINK(N : NODE; DEL : INTEGER);
      entry BREAK_LINK(N : NODE);
   end UPDATE_TABLE;

   task TABLE is
      entry READ(CURR_TABLE : out ALL_INFO);
      entry WRITE(NEW_TABLE : ALL_INFO;
                  NEW_ROUTE : ROUTE_INFO);
      entry GET_ROUTE(N : NODE; R : out EXTENDED_NODE);
   end TABLE;
```

```
task body TABLE is

   NODE_TABLE : ALL_INFO :=
        (ME => (DEL => 0, DIST => 0),
         others => (DEL => INFINITY, DIST => NODE'LAST));
   NODE_ROUTE : ROUTE_INFO :=
        (ME => ME, others => NULL_NODE);

begin  -- TABLE
   loop
      select
         accept READ(CURR_TABLE : out ALL_INFO) do
            CURR_TABLE := NODE_TABLE;
         end;
      or
         accept WRITE(NEW_TABLE : ALL_INFO;
                  NEW_ROUTE : ROUTE_INFO) do
            NODE_TABLE := NEW_TABLE;
            NODE_ROUTE := NEW_ROUTE;
         end;
      or
         accept GET_ROUTE(N : NODE; R : out EXTENDED_NODE) do
            R := NODE_ROUTE(N);
         end;
      end select;
   end loop;
end TABLE;

task body UPDATE_TABLE is
   HIS_TABLE : ALL_INFO;
   MY_TABLE  : ALL_INFO :=
            (ME => (DEL => 0, DIST => 0),
             others => (DEL => INFINITY, DIST => NODE'LAST));
   ROUTE : ROUTE_INFO :=
            (ME => ME, others => NULL_NODE);
   START_TIME : TIME;

begin -- UPDATE_TABLE
   loop
      START_TIME := SYSTEM'CLOCK;
      for I in NODE'FIRST .. NODE'LAST loop
         if NEIGHBOR(I) then
            LIBRARY.READ_NEIGHBOR_NODE(I, HIS_TABLE);
            for J in NODE'FIRST .. NODE'LAST loop
               if ROUTE(J) = I then
                  if HIS_TABLE(J).DIST < NODE'LAST then
                     MY_TABLE(J) :=
                        (DIST => HIS_TABLE(J).DIST + 1,
                         DEL  => HIS_TABLE(J).DEL +
                                 NEIGHBOR_DEL(I));
```

```
            else -- node J is unreachable
               MY_TABLE(J)  :=
                    (DIST => NODE'LAST,
                     DEL  => INFINITY);
               ROUTE(J)  := NULL_NODE;
            end if;
         elsif HIS_TABLE(J).DEL /= INFINITY and then
            MY_TABLE(J).DEL > HIS_TABLE(J).DEL +
                    NEIGHBOR_DEL(I) then
            -- establish a new ROUTE
            ROUTE(J)  := I;
            MY_TABLE(J)  :=
                    (DIST => HIS_TABLE(J).DIST + 1,
                     DEL  => HIS_TABLE(J).DEL +
                               NEIGHBOR_DEL(I));
         end if;
      end loop;
   end if;
end loop;

TABLE.WRITE(MY_TABLE, ROUTE);
select
   accept ESTABLISH_LINK(N : NODE; DEL : INTEGER) do
      NEIGHBOR(N)  := TRUE;
      NEIGHBOR_DEL(N)  := DEL;
      MY_TABLE(N)  := (DIST => 1, DEL => DEL);
      ROUTE(N)  := N;
   end;
or
   accept BREAK_LINK(N : NODE) do
      NEIGHBOR(N)  := FALSE;
      for I in NODE'FIRST .. NODE'LAST loop
         if ROUTE(I)  = N then
            MY_TABLE(I).DEL := INFINITY;
         end if;
      end loop;
   end;
end select;
delay START_TIME + BASIC_PERIOD - SYSTEM'CLOCK;
      end loop;
   end UPDATE_TABLE;

begin  -- MAIN
   initiate UPDATE_TABLE, TABLE;
end;
```

8. General Purpose Real-Time Scheduler

8.1 Sample Problem 8

Purpose:

An exercise to test the possibilities for relating computational processes to real time.

Problem:

A library module shall be written which allows to schedule computational processes in actual real time. The number of these processes shall be varying, determinable at link-time.

The scheduler shall receive the 'ticks' of the real-time clock of the system (e.g. by reacting to the respective interrupt) and transform them into actual real time, e.g. by applying the proper compile-time constants.

To simplify matters, the time span which can be handled by the scheduler may be restricted to 24 hours, i.e. all times will be computed modulo 24 hours.

This 'real time' shall be accessible to the program by the command

    TIME (OPERAND)

which shall deposit the time (at the point in time the operation is executed) in the location indicated by 'operand' as an ASCII character string with the following conventions:

    first two characters: hours
    second two characters: minutes
    third two characters: seconds

But the main purpose of the scheduler shall be the initiation of the execution of computational processes according to predefined conditions in real time. This shall be possible either once or repeatedly.

Processes shall be connected to the scheduler by operations of the form:

    EXECUTE PROCESSNAME, TIME
    EXECUTE TIME -- meaning the process which performs this operation

execute PROCESSNAME, START-TIME, REPETITION-INTERVAL

Intentionally, no exact representation for these operations is given in the example (especially it shall not be implied that they are procedure calls). The representation shall be proposed by the language designer in order to:

(1) fit into the text of a user program as simply and naturally as possible and

(2) be efficiently implementable in the language proposed.

If two processes are due for execution at the same point in time, they shall be activated in priority order.

Note, that in order to achieve this, a library routine may have to be used, which sorts the control blocks of the scheduled processes according to their priority. Because such a sorting routine is of general interest, it should also be useable for other data-types. It should be demonstrated, how the parameter passing mechanism of such a routine is fit for this purpose without causing too much runtime overhead.

For the purpose of the example, the sorting algorithm proper may be simple and inefficient, because it is not relevant for the demonstration.

It must also be possible to disconnect processes from the scheduler at any point in time, either by action from themselves or from other processes.

Assumptions:

Assume a system clock which delivers 'ticks' of a frequency which is sufficient to do the necessary computations with the necessary precision.

The way in which processes can be made known to the scheduler depends on the implementation model, which underlies the language proposal.

8.2 Green Solution to Problem 8

The real-time scheduler is implemented as a task (SCHEDULING_TASK), which is to run independently, and with a higher priority than all user processes. This task is defined inside the body of the package SCHEDULER.

A "user process" is any Green task which is "known" to the scheduler. In order to be known, the user process must contain a declaration of the form:

```
package MY_TASK is new SCHEDULER.TASK_PATTERN(PROCESS_NAME);
```

This will make the process known to the scheduler, under the corresponding name (given as a string). It will also make the three commands EXECUTE, DISCONNECT, and SIGN_OFF available to the user_process. Other user processes can be referred to in these commands by the name with which they have signed on, which may conveniently be the Green task name, although this is not a necessity.

<u>Meaning</u> <u>of</u> <u>Commands</u>:

```
EXECUTE(process_name, start_time [, period] );
```

requests execution of indicated process at given time, and optionally periodically thereafter. Note that, if the Green "keyword" notation is used, this command can be written:

```
EXECUTE(process_name, AT_TIME := start_time, THEN_EVERY := period);
```

The process_name can also be omitted, in which case, it is assumed to refer to the calling task, and will have the additional effect of suspending this task, after scheduling it to be resumed at the indicated time. The same effect will be achieved if the process_name that is explicitly given is actually that of the caller.

```
DISCONNECT(process_name);
```

This will cancel any previous scheduling request made for the indicated process.

```
SIGN_OFF
```

This command terminates the interactions of the calling task with the scheduler. It should be called before terminating a task (this is not a necessity, but avoids cluttering the name space of the scheduler).

Additional Rules Enforced

Several restrictions are enforced by the scheduler:

The same name cannot be used by more than one task.

A command cannot be executed if the calling task has not signed on, or has already signed off. If a task signs off, the only way to sign on again is by a new generic instantiation of TASK_PATTERN.

It is not possible to give a scheduling request for another user process that is still executing.

It is not possible to make a scheduling request for a process if one has already been made (although the effect can easily be achieved by the DISCONNECT command). However, when a task suspends itself, the start_time may be omitted, to use any previous scheduling coming from periodical execution request.

It is not possible to make a scheduling request for a time that has already elapsed.

Any violation of these rules raises an exception in the calling task.

Internals of the Scheduler

The scheduler is a constantly active task to which user processes are connected dynamically. There is a maximum number of user processes that can be connected to the scheduler at a given time. The connection is established by the generic instantiation of the package TASK_PATTERN. The initialization part of TASK_PATTERN will actually call the entry SIGN_ON of the scheduler. Note that all entries of the scheduler are not declared in the visible part, and can therefore be called only indirectly through the package TASK_PATTERN, which is nested in the scheduler.

Protection of the scheduler's integrity is achieved by the use of an internal process_name that cannot be manipulated or forged by user_processes: upon signing on, a variable local to the module part of TASK_PATTERN will be initialized by the scheduler, and thereafter used in each entry call to unambiguously designate the process. One of the effects of SIGN_OFF is to reset this variable to an inocuous value. A name table is maintained by the scheduler, to realize the mapping between process names and the internal indices used to refer to them.

In terms of scheduling, a central data structure, the delay list, is maintained, which contains information about the user processes that are to be scheduled. This list is sorted in chronological order, and, for a given due date, in the order in which the tasks have been suspended. This list only deals with suspended tasks for which a scheduling request has been made.

The flow of time is perceived by an entry TICK, possibly associated with a clock interrupt. The period of this clock is assumed to be sufficiently long, so that no tick is lost.

```
package SCHEDULER is

   MAX_TIME: constant INTEGER := 86400;
   type TIME is new INTEGER range 0 .. MAX_TIME;
   function CLOCK return TIME;

   LATE_REQUEST,    ALREADY_SCHEDULED, TOO_MANY_TASKS,
   INEXISTENT_TASK, TASK_STILL_ACTIVE, NOT_SIGNED_ON,
   ALREADY_SIGNED_ON: exception;

   generic (S: STRING)
   package TASK_PATTERN is
      procedure EXECUTE(WHO          : STRING;
                        AT_TIME      : TIME := 0;
                        THEN_EVERY   : TIME := 0);
      procedure EXECUTE(AT_TIME      : TIME;
                        THEN_EVERY   : TIME := 0);
      procedure DISCONNECT(WHO: STRING);
      procedure SIGN_OFF;
   end TASK_PATTERN;

end SCHEDULER;


package body SCHEDULER is

   -------------------------------------------------------------
   -- The task SCHEDULING_TASK is actually the real scheduler
   -------------------------------------------------------------
   task SCHEDULING_TASK is

      MAX_TASK: constant INTEGER := 63;
      type TASK_INDEX is new INTEGER range 0 .. MAX_TASK;
      subtype TASK_ID is TASK_INDEX  range 1 .. TASK_INDEX'LAST;

      function SEARCH_TASK(WHO : STRING) return TASK_ID;

      entry DISCONNECT(TSK: TASK_ID);
      entry REMOVE(TSK: TASK_ID);
      entry SCHEDULE(TSK, CALLER: TASK_ID; AT_TIME, THEN_EVERY: TIME);
      entry SIGN_ON(NAME: STRING; TSK: out TASK_ID);
      entry WAIT(TASK_ID'FIRST .. TASK_ID'LAST);

   end SCHEDULING_TASK;
```

```
----------------------------------------------------------------
-- The generic package TASK_PATTERN provides the user interface
-- with the scheduler. Each new instantiation makes a new
-- process known to the scheduler, by the call to SIGN_ON
-- done at initialization.
----------------------------------------------------------------
package body TASK_PATTERN is
   use SCHEDULING_TASK;
   ME: TASK_INDEX; -- ME is in fact a constant
                   -- which receives a value upon
                   -- package initialization.

   -----------------------------------------------------------
   -- EXECUTE corresponds to a scheduling request
   -- made for another process.
   -----------------------------------------------------------
   procedure EXECUTE(WHO        : STRING;
                     AT_TIME    : TIME;
                     THEN_EVERY : TIME) is
      WHOSE_ID: TASK_ID;
   begin
      if ME = 0 then
         raise NOT_SIGNED_ON;
      end if;
      WHOSE_ID := SEARCH_TASK(WHO);
      SCHEDULE(WHOSE_ID, ME, AT_TIME, THEN_EVERY);
      if WHOSE_ID = ME then
         WAIT(ME);
      end if;
   end EXECUTE;

   -----------------------------------------------------------
   -- This version of EXECUTE also suspends the caller
   -----------------------------------------------------------
   procedure EXECUTE(AT_TIME: TIME;
                     THEN_EVERY: TIME) is
   begin
      if ME = 0 then
         raise NOT_SIGNED_ON;
      else
         SCHEDULE(ME, ME, AT_TIME, THEN_EVERY);
         WAIT(ME);  --actually blocks the calling task
      end if;
   end EXECUTE;

   -----------------------------------------------------------
   -- To remove a scheduler entry
   -----------------------------------------------------------
   procedure DISCONNECT(WHO: STRING) is
   begin
      SCHEDULING_TASK.DISCONNECT(SEARCH_TASK(WHO));
   end DISCONNECT;
```

```
   -----------------------------------------------------
   -- To remove the process name from the scheduler's table
   -----------------------------------------------------
   procedure SIGN_OFF is
   begin
      if ME = 0 then
         raise NOT_SIGNED_ON;
      else
         REMOVE(ME);
         ME := 0;
      end if;
   end SIGN_OFF;

begin  --initialization of TASK_PATTERN
   SIGN_ON(S, ME);
end TASK_PATTERN;

-----------------------------------------------------------
-- BODY OF SCHEDULING_TASK
-----------------------------------------------------------
task body SCHEDULING_TASK is

   type TASK_NAME is access STRING;
   type TASK_STATUS is
      record
         ACTIVE, REQUESTED : BOOLEAN;
         START,  PERIOD    : TIME;
         NEXT,   PREVIOUS  : TASK_INDEX;
      end record;

   TIME_NOW: TIME := 0;
   DELAY_LIST: TASK_INDEX := 0;
   TIME_TABLE: array(1 .. MAX_TASK) of TASK_STATUS;
   NAME_TABLE: array(1 .. MAX_TASK) of TASK_NAME;

   procedure LINK(TSK: TASK_ID);
   procedure UNLINK(TSK: TASK_ID);

   entry TICK;

   for TICK use at -- interrupt address for tick;

   -----------------------------------------------------
   -- Find the internal id corresponding to a process name
   -----------------------------------------------------
   function SEARCH_TASK(WHO: STRING) return TASK_ID is
   begin
      for I in TASK_ID'FIRST .. TASK_ID'LAST loop
         if NAME_TABLE(I) /= null
            and then NAME_TABLE(I).all = WHO then
            return I;
         end if;
      end loop;
      raise INEXISTENT_TASK;
   end SEARCH_TASK;
```

```
function CLOCK return TIME is
begin
   return TIME_NOW;
end CLOCK;

-----------------------------------------------------------
-- Add a process in the delay list.
-- The process must have been already scheduled,
-- and must be blocked.
-----------------------------------------------------------
procedure LINK(TSK: TASK_ID) is
   I: TASK_ID;
   THIS: TASK_STATUS renames TIME_TABLE(TSK);
begin
   if DELAY_LIST = Ø then
      DELAY_LIST := TSK;
      THIS.PREVIOUS := Ø;
      THIS.NEXT := Ø;
   else
      I := DELAY_LIST;
      while TIME_TABLE(I).START <= THIS.START
         and TIME_TABLE(I).NEXT /= Ø loop
            I:= TIME_TABLE(I).NEXT;
      end loop;
      if TIME_TABLE(I).START > THIS.START then
         THIS.NEXT := I;
         THIS.PREVIOUS := TIME_TABLE(I).PREVIOUS;
         TIME_TABLE(I).PREVIOUS := TSK;
      else
         THIS.PREVIOUS := I;
         THIS.NEXT := TIME_TABLE(I).NEXT;
         TIME_TABLE(I).NEXT := TSK;
      end if;
   end if;
end LINK;

-----------------------------------------------------------
-- remove a process from the delay list
-----------------------------------------------------------
procedure UNLINK(TSK: TASK_ID) is
   THIS: TASK_STATUS renames TIME_TABLE(TSK);
begin
   if THIS.NEXT /= Ø then
      TIME_TABLE(THIS.NEXT).PREVIOUS := THIS.PREVIOUS;
   end if;
   if THIS.PREVIOUS /= Ø then
      TIME_TABLE(THIS.PREVIOUS).NEXT := THIS.NEXT;
      THIS.PREVIOUS := Ø;
   else    --first in delay_list
      DELAY_LIST := THIS.NEXT;
   end if;
   THIS.NEXT := Ø;
end UNLINK;
```

```
begin     -- body of scheduling_task
   loop
      declare     -- this inner block to catch all exceptions
         I: TASK_ID;
      begin
         select
            accept TICK;
            TIME_NOW := (TIME_NOW + 1) mod MAX_TIME;
            loop-- to release all processes due
                -- to be awoken now
               if DELAY_LIST /= 0
                  and then
                     TIME_TABLE(DELAY_LIST).START = TIME_NOW then
                     I := DELAY_LIST;
                     UNLINK(I);
                     TIME_TABLE(I).ACTIVE := TRUE;
                     if TIME_TABLE(I).PERIOD > 0
                        and then TIME_TABLE(I).START
                              + TIME_TABLE(I).PERIOD
                              <= MAX_TIME then
                           TIME_TABLE(I).START :=
                                 TIME_TABLE(I).START
                              + TIME_TABLE(I).PERIOD;
                           TIME_TABLE(I).REQUESTED := TRUE;
                     else
                        TIME_TABLE(I).REQUESTED := FALSE;
                     end if;
                     accept WAIT(I);-- actually release process
                                     -- with task-id I
               else exit;
               end if;
            end loop;
         or
            -- removes any previous request for TSK
            accept DISCONNECT(TSK: TASK_ID) do
               if TIME_TABLE(TSK).REQUESTED then
                  UNLINK(TSK);
               end if;
               TIME_TABLE(TSK).START := 0;
               TIME_TABLE(TSK).PERIOD := 0;
               TIME_TABLE(TSK).REQUESTED := FALSE;
            end DISCONNECT;
         or
            -- enter new request for TSK
            accept SCHEDULE(TSK, CALLER: TASK_ID;
                            AT_TIME, THEN_EVERY: TIME) do
               if TSK = CALLER then
                  TIME_TABLE(TSK).ACTIVE := FALSE;
               end if;
               if AT_TIME > 0 then
                  if AT_TIME < TIME_NOW then
                     raise LATE_REQUEST;
                  elsif TIME_TABLE(TSK).ACTIVE then
                     raise TASK_STILL_ACTIVE;
                  elsif TIME_TABLE(TSK).REQUESTED then
```

```
                    raise ALREADY_SCHEDULED;
                 else
                    TIME_TABLE(TSK).REQUESTED := TRUE;
                    TIME_TABLE(TSK).START := AT_TIME;
                    TIME_TABLE(TSK).PERIOD := THEN_EVERY;
                 end if;
              elsif TSK /= CALLER then
                 raise LATE_REQUEST;
              end if;
              if TIME_TABLE(TSK).REQUESTED then
                 LINK(TSK);
              end if;
           end SCHEDULE;
        or
           accept REMOVE(TSK: TASK_ID) do
              NAME_TABLE(TSK) := null;
           end REMOVE;
        or
           -- enter new name in name table
           -- and allocate corresponding task_id
           accept SIGN_ON(NAME: STRING; TSK: out TASK_ID) do
              for I in TASK_ID'FIRST .. TASK_ID'LAST loop
                 if NAME_TABLE(I) = null then
                    NAME_TABLE(I) := new TASK_NAME(NAME);
                    TSK := I;
                    TIME_TABLE(I) := (FALSE, FALSE, 0, 0, 0, 0);
                    return;
                 elsif NAME_TABLE(I).all = NAME then
                    raise ALREADY_SIGNED_ON;
                 end if;
              end loop;
              raise TOO_MANY_TASKS;
           end SIGN_ON;
        end select;
     exception
        when others =>
           null;
     end;
  end loop;
end SCHEDULING_TASK;

begin  -- initialization of SCHEDULER
   initiate SCHEDULING_TASK;
end SCHEDULER;
```

9. Distributed Parallel Output

9.1 Sample Problem 9

Purpose:

An exercise to demonstrate the ability of processing parallel events which need not progress at the same rate.

Problem:

This program has encountered a multiple addressee message to be output over a number of asynchronous links.

Each link is controlled by an individual process which performs all link related processing. Each process can accept one packet of the message at a time and will notify the program when the last packet furnished to it has been acknowledged by the distant station.

When all transmissions are complete, the program shall purge the message.

Assumptions:

(1) The message has five addressees, but these can be different for each message.

(2) The message is five packets long.

(3) Each packet is 8Ø bytes long.

(4) The buffers containing the message are contiguously located.

(5) At initialization, the program shall be furnished the address of the first buffer, the number of buffers, and the identity of the five links over which the message is to be sent (each link is controlled by an individual process, named LØ..L9).

The link identification shall be in the form (Ln, Ln, Ln...) where n has legal values between Ø and 9.

(6) An 8 bit machine (one of today's typical microprocessors)

(7) The program will be capable of processing up to ten addressees.

(8) There is no queuing delay, i.e. the link-processes are dedicated and can react immediately.

Remark: One can assume that the individual link processes are resident in dedicated microprocessors and that the coordination is done in another processor to which they are connected by a bus.

Guidelines:

None.

9.2 Green Solution to Problem 9

In order to achieve the desired degree of parallelism, it is necessary that a message transmission can be started before previous ones have been completed. In addition, the links should not be compelled to transmit contiguous packets of the same message: the transmission of packets of different messages over a given link can be skewed (reassembly is assumed to take place at the other end).

To these ends, a task, MESSAGE, repeatedly accepts transmission requests. Since there is a maximum number of buffers in the system, and a fixed number of addressees per message, there is a maximum number of message transmissions that can be requested at a given time. For each message to be transmitted over a particular link, a member of the task family PACKETIZER is initiated: it will successively forward all the packets of the message on the appropriate link.

Each link is driven by a LINK_CONTROLLER which repeatedly accepts a packet and sends it. It can also receive a distant acknowledgement which identifies the message received. This causes an acknowledgement to be forwarded to MESSAGE. When MESSAGE has received five acks for the same message, it can release the corresponding buffers.

For the sake of simplicity, we have adopted a fixed message size. Messages are identified by the index of their first buffer, rather than by a complex sequence number. Each buffer is guarded by an AVAILABLE flag. Resetting the flag has the effect of releasing the buffer, thus purging its contents.

```
package MESSAGE_TRANSMISSION is

    N_LINKS    : constant INTEGER := 10;
    PKT_SIZE   : constant INTEGER := 80;
    MSG_SIZE   : constant INTEGER := 5;
    N_BUFFERS  : constant INTEGER := 100;

    type LINK      is new INTEGER range 0 .. N_LINKS-1;
    type DEST_SET  is array (1 .. 5) of LINK;
    type PACKET    is array (1 .. PKT_SIZE) of CHARACTER;
    subtype MSG_ID is INTEGER range 1 .. N_BUFFERS;

    BUFFERS   : array(MSG_ID'FIRST .. MSG_ID'LAST) of PACKET;
    AVAILABLE : array(MSG_ID'FIRST .. MSG_ID'LAST) of BOOLEAN
                := (MSG_ID'FIRST .. MSG_ID'LAST => TRUE);

    task MESSAGE is
        entry SEND (MSG : MSG_ID; TO   : DEST_SET);
        entry ACK  (MSG : MSG_ID; FROM : LINK);
    end;
end;

package body MESSAGE_TRANSMISSION is

    task PACKETIZER (1 .. 100) is
        entry SEND (MSG : MSG_ID; TO : LINK);
    end;

    task LINK_CONTROLLER (LINK'FIRST .. LINK'LAST) is
        entry SEND (PKT, MSG : MSG_ID);
        entry ACK  (MSG : MSG_ID);
    end;

    task body PACKETIZER is
        M    : MSG_ID;
        DEST : LINK;
    begin
        accept SEND (MSG : MSG_ID; TO : LINK) do
            M := MSG;
            DEST := TO;
        end SEND;

        for I in 1 .. MSG_SIZE loop
            LINK_CONTROLLER(DEST).SEND(M+I-1, M);
        end loop;
    end PACKETIZER;
```

```
    task body LINK_CONTROLLER is
    begin
       loop
          select
             accept SEND (PKT, MSG : MSG_ID) do
             -- transmit packet over the link
             end SEND;

          or accept ACK (MSG : MSG_ID) do
                MESSAGE.ACK(MSG, LINK_CONTROLLER'INDEX);
             end ACK;
          end select;
       end loop;
    end LINK_CONTROLLER;

    task body MESSAGE is
       ACK_COUNT : array (MSG_ID'FIRST .. MSG_ID'LAST) of INTEGER
                         := (MSG_ID'FIRST .. MSG_ID'LAST => 0);
    begin
       loop
          select
             accept SEND (MSG : MSG_ID; TO : DEST_SET) do
                for I in 1 .. MSG_SIZE loop
                   initiate PACKETIZER((MSG-1)*MSG_SIZE + I);
                   PACKETIZER((MSG-1)*MSG_SIZE + I).SEND(MSG, TO(I));
                end loop;
             end SEND;

          or accept ACK (MSG : MSG_ID; FROM : LINK) do
                ACK_COUNT(MSG) := ACK_COUNT(MSG) + 1;
                if ACK_COUNT(MSG) = MSG_SIZE then
                   ACK_COUNT(MSG) := 0;
                   for I in 1 .. MSG_SIZE loop
                      AVAILABLE(MSG + I - 1) := TRUE;
                   end loop;
                end if;
             end ACK;
          end select;
       end loop;
    end MESSAGE;

 begin  -- body of MESSAGE_TRANSMISSION
    initiate MESSAGE, LINK_CONTROLLER(LINK'FIRST .. LINK'LAST);
 end MESSAGE_TRANSMISSION;
```

10. Unpacking and Conversion of Data

10.1 Sample Problem 10

Purpose:

An exercise to process a packed binary message header.

Problem:

A packed binary message packet has been received and placed in a buffer by the line handler. This program's task is to determine the classification, precedence and destination of the message packet. These data shall then be reordered and placed in a queue entry for later processing by another program.

Assumptions:

1 An 8 bit machine (one of today's typical microprocessors)

2 The buffer is assigned from a buffer pool. The exact location of the buffer is supplied to the program when it is invoked.

3 The packet may be up to 256 bytes long.

4 The packet-format is:

```
byte 0 - bits 0-2 : classification 0 top-secret
                                   1 secret
                                   2 confidential
                                   3 unclass
                          all others unknown

         bits 3-4 : precedence     0 routine
                                   1 priority
                                   2 flash
                                   3 unknown

         bits 5-7   and byte 1 - bits 0-7 : addressee

byte 2 - bits 0-7 : packet length in bytes
```

5 Queue entry format

byte Ø -classification ascii characters (!)

T - top-secret
S - secret
C - confidential
U- unclassified
X - unknown

byte 1 -precedence

F - flash
P - priority
R - routine
X - unknown

byte 2 and 3 - Addressee (right justified, not converted)

byte 4 -packet length

byte 5 -packet number

6 Queue entries are obtained from a common pool by calling the routine GET_A_QUEUE..

7 To simplify matters, assume an infinite supply of queue entries.

8 A packet is passed to a program prior to this routine's termination.

Guidelines

None.

1Ø.2 Green Solution to Problem 1Ø

The solution uses two record type declarations, PACKET_FORMAT and QE_FORMAT, with adequate representation specifications. The unpacking and conversion of addressee and packet length are just a matter of component assignment. The conversions of classification and precedence must utilize a case statement, as the representations for the corresponding enumeration types are not ordered in the same way.

```
restricted(LINE_HANDLER, QUEUE_HANDLER)
task PACKET_TO_QUEUE is

   BYTE       : constant INTEGER := 8;
   MAX_PACKET : constant INTEGER := 255;
   type PACKET_NUMBER is new INTEGER range 0 .. MAX_PACKET;

   package PACKET_TYPE is
      MAX_LENGTH : constant INTEGER := 256;

      type CLASSIFICATION is
               (TOP_SECRET, SECRET,     CONFIDENTIAL, UNCLASSIFIED,
                UNKNOWN_4,  UNKNOWN_5,  UNKNOWN_6,    UNKNOWN_7);

      type PRECEDENCE is
                (ROUTINE, PRIORITY, FLASH, UNKNOWN);

      type ADDRESS is new INTEGER range 0 .. 1023;
      type LENGTH  is new INTEGER range 0 .. MAX_LENGTH;

      type PACKET_FORMAT is
         record
            CLASS : CLASSIFICATION;
            PREC  : PRECEDENCE;
            DEST  : ADDRESS;
            LGTH  : LENGTH;
         end record;

      for CLASSIFICATION use
         (TOP_SECRET   => 0,
          SECRET       => 1,
          CONFIDENTIAL => 2,
          UNCLASSIFIED => 3,
          UNKNOWN_4    => 4,
          UNKNOWN_5    => 5,
          UNKNOWN_6    => 6,
          UNKNOWN_7    => 7);

      for PRECEDENCE use
         (ROUTINE  => 0,
          PRIORITY => 1,
          FLASH    => 2,
          UNKNOWN  => 3);

      for PACKET_FORMAT use
         record
            CLASS at 0*BYTE range 0 .. 2;
            PREC  at 0*BYTE range 3 .. 4;
            DEST  at 0*BYTE range 5 .. 15;
            -- note that bit positions may exceed word size
            LGTH  at 2*BYTE range 0 .. 7;
         end record;

   end PACKET_TYPE;
```

```
      package QUEUE_ENTRY_TYPE is
         type CLASSIFICATION is new CHARACTER;
         type PRECEDENCE     is new CHARACTER;
         subtype ADDRESS     is PACKET_TYPE.ADDRESS;
         subtype LENGTH      is PACKET_TYPE.LENGTH;
         type QE_FORMAT is
            record
               CLASS : CLASSIFICATION;
               PREC  : PRECEDENCE;
               DEST  : ADDRESS;
               LGTH  : LENGTH;
               NBR   : PACKET_NUMBER;
            end record;
         for QE_FORMAT use
            record
               CLASS at 0*BYTE range 0 .. 7;
               PREC  at 1*BYTE range 0 .. 7;
               DEST  at 2*BYTE range 0 .. 15;
               LGTH  at 4*BYTE range 0 .. 7;
               NBR   at 5*BYTE range 0 .. 7;
            end record;
      end QUEUE_ENTRY_TYPE;

      PKT_INDEX : PACKET_NUMBER;
      CUR_PKT   : PACKET_TYPE.PACKET_FORMAT;
      CUR_QE    : QUEUE_ENTRY_TYPE.QE_FORMAT;

   begin
      loop
         declare
            use PACKET_TYPE, QUEUE_ENTRY_TYPE;
         begin
            LINE_HANDLER.GET_A_PACKET(PKT_INDEX);
            CUR_PKT := LINE_HANDLER.BUFFER_POOL(PKT_INDEX);
            QUEUE_HANDLER.GET_A_QUEUE(CUR_QE);
            case CUR_PKT.CLASS of
               when TOP_SECRET   => CUR_QE.CLASS := "T";
               when SECRET       => CUR_QE.CLASS := "S";
               when CONFIDENTIAL => CUR_QE.CLASS := "C";
               when UNCLASSIFIED => CUR_QE.CLASS := "U";
               when others       => CUR_QE.CLASS := "X";
            end case;
            case CUR_PKT.PREC of
               when ROUTINE  => CUR_QE.PREC := "R";
               when PRIORITY => CUR_QE.PREC := "P";
               when FLASH    => CUR_QE.PREC := "F";
               when others   => CUR_QE.PREC := "X";
            end case;
            CUR_QE.DEST := CUR_PKT.DEST;
            CUR_QE.LGTH := CUR_PKT.LGTH;
            CUR_QE.NBR  := PKT_INDEX;
            QUEUE_HANDLER.SEND_QUEUE(CUR_QE);
         end;
      end loop;
   end PACKET_TO_QUEUE;
```