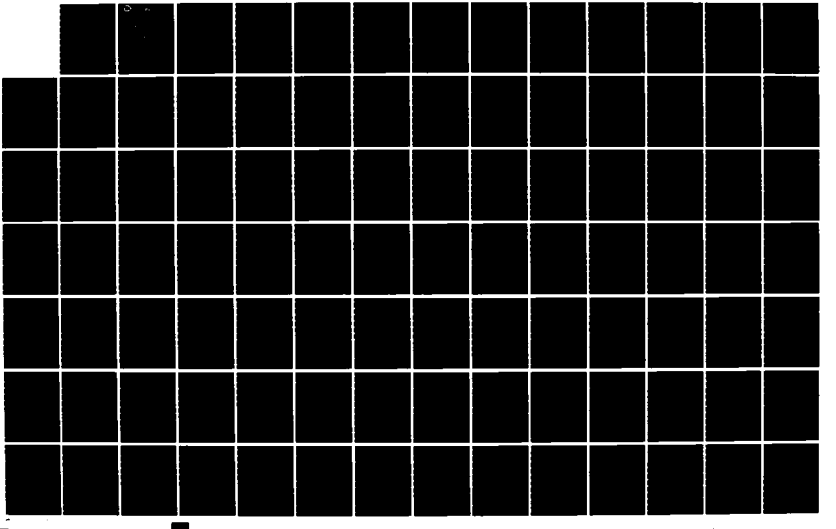
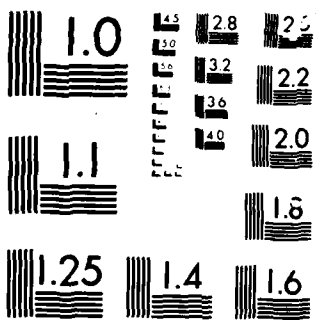
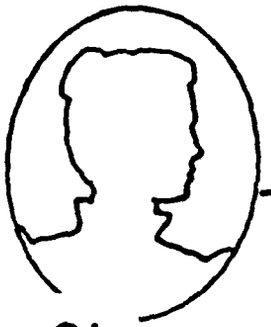
AD-A166 352 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS IN ADA L401 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 1/6

UNCLASSIFIED F/G 5/9 NL





MICROCOPY RESOLUTION TEST CHART



AD-A166 352

1986

Ada® Training Curriculum

Real-Time Systems In Ada
L401
Teacher's Guide
Volume II

DTIC
ELECTE
APR 03 1986
S D
E

Superseded
AD-A146783

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K506
14

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

86 4 3 043

•Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

*Approved For Public Release/Distribution Unlimited

PART IV

FUNDAMENTAL TASK DESIGNS

11. SERVER AND USER TASKS

12. MONITORS

13. MESSAGE BUFFERS

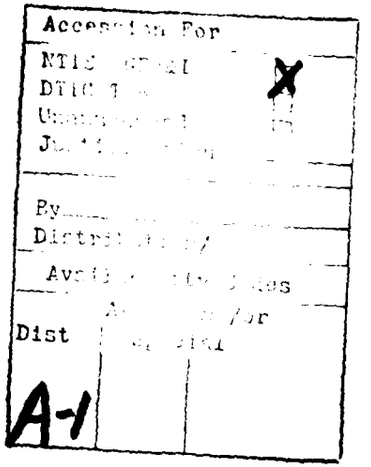
14. CYCLIC PROCESSING

15. STREAM-ORIENTED TASK DESIGN

INSTRUCTOR NOTES

ALLOW 30 MINUTES FOR THIS SECTION.

RECOMMEND THAT STUDENTS READ EXERCISE 4.1 IN THE REAL-TIME ADA WORKBOOK.



Section 11

SERVER AND USER TASKS

INSTRUCTOR NOTES

THIS SLIDE GIVES THE MAIN MESSAGE OF SECTION 11.

- BULLET 2: THE EXAMPLE WAS FIRST INTRODUCED IN SECTION 8.

- BULLET 3: THE NEXT TWO SLIDES PURSUE THE ANALOGY BETWEEN PROCEDURES AND ENTRIES.

TASKS PROVIDE SERVICES THROUGH ENTRIES

- WHEN TASK A CALLS AN ENTRY OF TASK B, WE CAN USUALLY VIEW TASK B AS PROVIDING SOME SERVICE TO TASK A.
  - TASK A REQUESTS THE SERVICE BY CALLING THE ENTRY.
  - TASK B PROVIDES THE SERVICE BY EXECUTING AN ACCEPT STATEMENT FOR THE ENTRY.
- A TASK'S ENTRIES CORRESPOND TO THE SERVICES THAT A TASK PROVIDES.

```
task type Shared_Count_Type is
   entry Increase_Count (By : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);
end Shared_Count_Type;
```

- FROM THE CALLING TASK'S POINT OF VIEW, AN ENTRY CALL IS VERY MUCH LIKE A PROCEDURE CALL.
  - CONTEXT:

```
Event_Count : Shared_Count_Type;
N           : Integer;
```

  - EXAMPLES:

```
Event_Count.Increase_Count (4);
Event_Count.Get_Count (N);
```

  - EXECUTION OF A CALL STATEMENT CAUSES SOME SERVICE TO BE PERFORMED.
  - THE SERVICE TO BE PERFORMED MAY BE SPECIFIED BY <u>in</u> OR <u>in out</u> PARAMETERS.
  - PERFORMANCE OF THE SERVICE MAY BE REFLECTED IN THE SETTING OF <u>in out</u> OR <u>out</u> PARAMETERS.

INSTRUCTOR NOTES

- BULLET 1: THE CONCEPTUAL SIMILARITY WAS EXPLAINED ON THE PREVIOUS SLIDE.

- BULLET 2:

  - ITEM 3:

    PARAMETER-TYPE PROFILES AND OVERLOADING WERE EXPLAINED IN MODULE L305. ESSENTIALLY, A PARAMETER-TYPE PROFILE OF A PROCEDURE OR ENTRY IS AN ORDERED LIST OF THE BASE TYPES OF THE PARAMETERS. (FUNCTIONS HAVE PARAMETER/RESULT TYPE PROFILES.)

    THE EXAMPLE ASSUMES THAT A SYSTEM HAS EVOLVED OVER THE YEARS TO INCLUDE BOTH OLD AND NEW MODELS OF A CERTAIN KIND OF SENSOR. THE NEW MODEL PROVIDES MORE PRECISE DATA. THERE ARE FIXED-POINT TYPES DECLARED FOR THE DATA PROVIDED BY EACH MODEL.

    EACH SENSOR HAS A TASK THAT CALLS Data_Analysis_Task.Report_Sensor_Reading WITH EACH PIECE OF DATA IT OBTAINS. THERE ARE ACTUALLY TWO SEPARATE ENTRIES BY THAT NAME, AND THE TYPE OF THE ACTUAL PARAMETER DETERMINES WHICH ONE IS CALLED. SIMILARLY, Data_Analysis_Task SHOULD HAVE AT LEAST TWO ACCEPT STATEMENTS FOR Report_Sensor_Reading, ONE WITH EACH FORMAL PARAMETER TYPE. THE TYPE OF THE FORMAL PARAMETER DETERMINES WHICH Report_Sensor_Reading ENTRY THE ACCEPT STATEMENT APPLIES TO.

    OVERLOADING ALLOWS TASKS FOR BOTH KINDS OF SENSORS TO BE WRITTEN IN THE SAME WAY.

  - ITEM 4:

    THE NEXT SLIDE ADDRESSES ENTRIES AS PROCEDURES IN RENAMING DECLARATIONS. AN ENTRY MAY BE USED AS A GENERIC ACTUAL PARAMETER CORRESPONDING TO A GENERIC FORMAL PROCEDURE WITH MATCHING PARAMETER TYPES AND MODES.

SIMILARITIES BETWEEN ENTRIES AND PROCEDURES

- CONCEPTUAL SIMILARITY:
  - ENTRY CALLS AND PROCEDURE CALLS ARE BOTH WAYS TO OBTAIN A SERVICE.
- STRUCTURAL SIMILARITIES:
  - SIMILAR CALLING SYNTAX
    - ONLY DIFFERENCE IS HOW THE SERVER IS NAMED:

      `procedure name` [(`actual parameter list`)];

      `task name`.`entry name` [(`actual parameter list`)];

    - ACTUAL PARAMETER LIST MAY BE NAMED, POSITIONAL, OR MIXED.
  - SAME PARAMETER MECHANISM
    - ACTUAL PARAMETERS IN USER CORRESPOND TO FORMAL PARAMETERS IN SERVER
    - FORMAL PARAMETERS HAVE MODE in, out, OR in out.
    - PARAMETERS WITH MODE in MAY BE GIVEN DEFAULT VALUES:

```
entry Increase_Count (By : in Positive := 1);
-- The call Event_Count.Increase_Count is equivalent to
-- Event_Count.Increase_Count (1);
```

  - EITHER KIND OF CALL MAY BE FINISHED BY EXECUTION OF A return STATEMENT
  - SAME OVERLOADING RULES
    - ENTRIES WITH DIFFERENT PARAMETER-TYPE "PROFILES" MAY BE OVERLOADED:

```
type Model_1_Sensor_Reading_Type is delta 0.1 range 0.0 .. 1500.0;
type Model_2_Sensor_Reading_Type is delta 0.01 range 0.0 .. 2.0E8;
task Data_Analysis_Task is
   entry Report_Sensor_Reading (Data : in Model_1_Sensor_Reading_Type);
   entry Report_Sensor_Reading (Data : in Model_2_Sensor_Reading_Type);
   ...
end Data_Analysis_Task;
```

  - ENTRIES MAY BE USED AS PROCEDURES IN RENAMING DECLARATIONS AND GENERIC INSTANTIATIONS.

INSTRUCTOR NOTES

THE SYNTAX FOR RENAMING AN ENTRY AS A PROCEDURE IS THE SAME AS FOR RENAMING A PROCEDURE AS A PROCEDURE. THE RENAMING CAN SPECIFY NEW PARAMETER NAMES AND DEFAULT VALUES, BUT THE NUMBER AND TYPE OF PARAMETERS MUST MATCH.

RENAMING MAY CAUSE OVERLOADING.

BULLETS 2 AND 3 GIVE TWO REASONS FOR RENAMING ENTRIES: SUCCINCTNESS AND ABSTRACTION.

- BULLET 5:
  - ITEM 1:

    A TASK BODY IS THE TASK BODY FOR ALL OBJECTS IN ITS TASK TYPE.

ENTRIES AS PROCEDURES IN REMANING DECLARATIONS

- CONTEXT:

```
task Message_Buffer is
    entry Send_Message (Message : in Message_Type);
    entry Receive_Message (Message : out Message_Type);
end Message_Buffer;
```

- TO PROVIDE SIMPLE NAMES FOR ENTRIES:

```
procedure Send_Message (Message : in Message_Type)
    renames Message_Buffer.Send_Message;

procedure Receive_Message (Message : out Message_Type)
    renames Message_Buffer.Receive_Message;
```

- TO PROVIDE AN ABSTRACT VIEW, HIDING THE FACT THAT SOME SERVICE IS PROVIDED BY A TASK:

```
procedure Write (Message : in Message_Type)
    renames Message_Buffer.Send_Message;

procedure Read (Message : out Message_Type)
    renames Message_Buffer.Receive_Message;
```

- A CALL ON THE RENAMED ENTRY LOOKS LIKE A PROCEDURE CALL:
  - SYNTACTIC FORM
  - EFFECT OF OBTAINING SOME SERVICE

- NEW NAME NOT USED INSIDE THE CALLED TASK'S TASK BODY.
  - ONLY ONE TASK OBJECT'S ENTRY IS RENAMED, BUT TASK BODY MAY CORRESPOND TO SEVERAL TASK OBJECTS.
  - CALLED TASK CANNOT DISTINGUISH BY WHICH NAME ITS ENTRY IS CALLED.

INSTRUCTOR NOTES

- BULLET 1:

  - ITEM 2:

    AN ACCEPT STATEMENT DOES NOT REFER IN ANY WAY TO THE TASK CALLING THE ENTRY. (OF COURSE ONE OF THE ENTRY'S PARAMETERS COULD CONTAIN INFORMATION IDENTIFYING THE CALLER.)

- BULLET 3:

  BECAUSE CALLING TASKS ARE USERS, THEY REQUEST A SPECIFIC SERVICE FROM A SPECIFIC TASK. CALLED TASKS CAN FULFILL REQUESTS FOR SERVICE WITHOUT REGARD TO WHO MADE THE REQUEST.

  FURTHERMORE, A USER TASK GENERALLY NEEDS A SPECIFIC SERVICE PERFORMED AT A PARTICULAR POINT IN ITS ALGORITHM BEFORE IT CAN PROCEED, WHILE A SERVER TASK CAN WAIT TO SERVE WHICHEVER REQUEST ARRIVES NEXT.

RENDEZVOUS ARE ASYMMETRIC

- KNOWLEDGE OF THE OTHER PARTY TO THE RENDEZVOUS
  - A CALLING TASK NAMES THE TASK WHOSE ENTRY IT IS CALLING:

```
Event_Count, Response_Count : Shared_Count_Type;
   ...
Event_Count.Increase_Count (N);
Response_Count.Increase_Count (1);
```

  - A CALLED TASK HAS NO MECHANISM FOR DETERMINING WHICH TASK ISSUED THE ENTRY CALL IT IS ACCEPTING.

```
accept Increase_Count (By : in Positive) do
   Sum := Sum + By;
end Increase_Count;
```

- WAITING FOR A RENDEZVOUS
  - A TASK CAN WAIT TO ACCEPT A CALL ON WHICHEVER OF SEVERAL ENTRIES IS CALLED FIRST:

```
select
   accept Increase_Count (By : in Positive) do
      Sum := Sum + By;
   end Increase_Count;
or
   accept Get_Count (Sum_So_Far : out Natural) do
      Sum_So_Far := Sum;
   end Get_Count;
end select;
```

  - A TASK CAN ONLY CALL ONE ENTRY AT A TIME.
- REASON FOR DIFFERENCES:
  - CALLED TASKS ARE "SERVERS"
  - CALLING TASKS ARE "USERS"

INSTRUCTOR NOTES

A SPECIFIC EXAMPLE FOLLOWS THIS SLIDE.

- BULLET 2:
  OFTEN, EITHER TASK COULD JUSTIFIABLY BE SEEN AS SERVING THE OTHER. THE TASK DESIGN DEPENDS ON WHICH VIEWPOINT IS CHOSEN.
- BULLET 3:
  - ITEM 1:
    A USER TASK MUST EXPLICITLY NAME EACH TASK IT CALLS, BUT A SERVER TASK ACCEPTS CALLS FROM WHOEVER ISSUES THEM.
  - ITEM 2:
    A USER TASK ISSUES CALLS IN A PARTICULAR ORDER, BUT A SERVER TASK CAN CONTAIN A SELECTIVE WAIT ACCEPTING CALLS ON DIFFERENT ENTRIES AS THOSE CALLS ARE ISSUED.
  - ITEM 3:
    A SERVER TASK TYPICALLY WAITS IN A SELECT STATEMENT UNTIL SOME REQUEST FOR SERVICE ARRIVES, WHILE A USER TASK REQUIRES SERVICE AT A PARTICULAR POINT IN ITS PROCESSING. THUS THE USER TASK INITIATES THE RENDEZVOUS BY CALLING AN ENTRY.
  - ITEM 4
    AN accept STATEMENT CAN ONLY APPEAR DIRECTLY IN THE SEQUENCE OF STATEMENTS OF A TASK BODY, NOT IN A SUBPROGRAM. AN ENTRY CALL CAN APPEAR IN ANY SEQUENCE OF STATEMENTS. IF THE INTERACTION OCCURS IN SEVERAL PLACES, SURROUNDED BY THE SAME STATEMENTS IN EACH PLACE, THOSE STATEMENTS ARE AN APPROPRIATE CANDIDATE FOR A PROCEDURE BODY.

REVERSING THE DIRECTION OF RENDEZVOUS

DESIGN HINT

SOMETIMES THE DESIGN OF A MULTITASK PROGRAM CAN BE SIMPLIFIED BY REVERSING THE DIRECTION OF RENDEZVOUS BETWEEN TWO TASKS.

- INSTEAD OF TASK A CALLING AN ENTRY OF TASK B, TASK B CALLS AN ENTRY OF TASK A.
- THIS ENTAILS RECONSIDERING WHICH TASK SERVES WHICH WHEN THE TWO TASKS COMMUNICATE.
- ISSUES TO CONSIDER
  - WILL ONE TASK BE INTERACTING IN THE SAME WAY WITH MANY DIFFERENT TASKS? (YES => SERVER)
  - WILL THE TASK BE PERFORMING DIFFERENT KINDS OF INTERACTIONS IN AN UNPREDICTABLE ORDER? (YES => SERVER)
  - DOES THE TASK TAKE THE INITIATIVE IN DETERMINING WHEN AN INTERACTION SHOULD TAKE PLACE? (YES => USER)
  - IS THE INTERACTION MOST CONVENIENTLY PERFORMED FROM WITHIN A SUBPROGRAM? (YES => USER)

INSTRUCTOR NOTES

THIS EXAMPLE WAS INTRODUCED IN SECTION 2.

THE Heater_Task BODY IS GIVEN ON THE NEXT SLIDE.

THIS SOLUTION WILL BE IMPROVED UPON SHORTLY BY REVERSING THE DIRECTION OF THE RENDEZVOUS.

EXAMPLE: MULTIZONE HEATING SYSTEM

- A TASK CONTROLLING THE HEATER SETS THE HEATER AT Off, Low, Medium, OR High, DEPENDING ON THE NUMBER OF OPEN VENTS.
- TASKS FOR EACH ZONE SERVE THE HEATER TASK BY PROVIDING THEIR CURRENT STATUS (Opened OR Closed) WHEN THEIR Get_Vent_Status ENTRIES ARE CALLED.
- DECLARATIONS:

```
Number_Of_Zones:  constant := 5;

type Zone_ID_Type is range 1 .. Number_Of_Zones;
type Vent_Status_Type is (Closed, Opened);

task Heater_Task is
      ...
end Heater_Task;

task type Zone_Task_Type is
      ...
   entry Get_Vent_Status (Vent_Status:  out Vent_Status_Type);
      ...
end Zone_Task_Type;

Zone_Task_List:  array (Zone_ID_Type) of Zone_Task_Type;
```

INSTRUCTOR NOTES

THE TASK BODY KEEPS TRACK OF THE CURRENT STATUS OF EACH VENT IN Vent_Status_List.

THE OUTER LOOP REPEATEDLY INVOKES THE INNER LOOP. THE INNER LOOP PULLS THE CURRENT STATUS OF EACH OF THE VENTS IN TURN, COMPARING IT WITH THE STATUS IN Vent_Status_List. IF THE STATUS OF THE VENT HAS CHANGED SINCE IT WAS LAST PULLED, Vent_Status_List AND Open_Vent_Count ARE UPDATED ACCORDINGLY AND THE HEATER SETTING IS READJUSTED IF NECESSARY.

Required_Setting IS A TABLE GIVING THE APPROPRIATE HEATER SETTING FOR ANY NUMBER OF OPEN VENTS.

Heater_Task BODY

```
task body Heater_Task is

      type Heater_Setting_Type is (Off, Low, Medium, High);
      Current_Heater_Setting : Heater_Setting_Type := Off;
      Required_Setting       : constant array (0 .. Number_Of_Zones)
                                  of Heater_Setting_Type :=
                                     (Off, Low, Medium, Medium, High, High);
      Open_Vent_Count        : Integer range 0 .. Number_Of_Zones := 0;
      Vent_Status_List       : array (Zone_ID_Type) of Vent_Status_Type :=
                                  (Zone_ID_Type => Closed);
      New_Status             : Vent_Status_Type;
            ...

begin

      loop
         for Zone in Zone_ID_Type loop
             Zone_Task_List (Zone).Get_Vent_Status (New_Status);
             if New_Status /= Vent_Status_List (Zone) then
                Vent_Status_List (Zone) := New_Status;
                case New_Status is
                    when Opened =>
                       Open_Vent_Count := Open_Vent_Count + 1;
                    when Closed =>
                       Open_Vent_Count := Open_Vent_Count - 1;
                end case;
                if Required_Setting (Open_Vent_Count) /= Current_Heater_Setting then
                    Current_Heater_Setting := Required_Setting (Open_Vent_Count);
                    Set_Heater (Current_Heater_Setting);
                end if;
             end if;
         end loop;
      end loop;

end Heater_Task;
```

INSTRUCTOR NOTES

- BULLET 1:

  IN THE FIRST VIEW, THE ZONE TASKS SERVED THE HEATER TASKS BY GIVING THE CURRENT STATUS OF THEIR VENTS WHEN REQUESTED; THE HEATER TASK REQUESTED AND THEN ACTED UPON THIS INFORMATION. IN THE SECOND VIEW, THE HEATER TASK SERVES THE ZONE TASKS BY ACTING UPON A REPORT THAT A VENT HAS OPENED OR CLOSED; A ZONE TASK REQUESTS SUCH ACTION WHEN IT OPENS OR CLOSES A VENT.

- BULLET 3:

  IT IS EASY TO DETERMINE WHERE IN THE Zone_Task_Type BODY THE ENTRY CALLS SHOULD GO. A ZONE TASK SHOULD CALL Report_Vent_Open WHEN IT OPENS A VENT AND Report_Vent_Closed WHEN IT CLOSES A VENT.

- BULLET 4:

  THE REVISED HEATER TASK REPEATEDLY WAITS FOR A CALL ON Report_Vent_Open OR Report_Vent_Closed, ADJUSTS Open_Vent_Count BASED ON WHICH ENTRY WAS ACCEPTED, AND THEN ADJUSTS THE HEATER SETTING AS REQUIRED.

A BETTER VIEW

- THE HEATER TASK SERVES THE ZONE TASKS, BY HANDLING REPORTS THAT A VENT HAS BEEN OPENED OR CLOSED.
- Heater_Task IS GIVEN TWO ENTRIES:

```
entry Report_Vent_Open;
entry Report_Vent_Closed;
```

- Zone_Task_Type BODY CALLS Heater_Task.Report_Vent_Open UPON OPENING A VENT AND Heater_Task.Report_Vent_Closed UPON CLOSING A VENT.
- Heater_Task BODY:

```
task body Heater_Task is
    type Heater_Setting_Type is (Off, Low, Medium, High);
    Current_Heater_Setting : Heater_Setting_Type := Off;
    Required_Setting       : constant array (0 .. Number_Of_Zones)
                                 of Heater_Setting_Type :=
                                    (Off, Low, Medium, Medium, High, High);
    Open_Vent_Count        : Integer range 0 .. Number_Of_Zones := 0;
       ...
begin
    loop
       select
           accept Report_Vent_Open;
           Open_Vent_Count := Open_Vent_Count + 1;
       or
           accept Report_Vent_Closed;
           Open_Vent_Count := Open_Vent_Count - 1;
       end select;
       if Required_Setting (Open_Vent_Count) /= Current_Heater_Setting then
           Current_Heater_Setting := Required_Setting (Open_Vent_Count);
           Set_Heater (Current_Heater_Setting);
       end if;
    end loop;
end Heater_Task;
```

INSTRUCTOR NOTES

THE FIRST TWO BULLETS ARE CONCERNED WITH PERFORMANCE, THE LAST THREE WITH SIMPLICITY OF THE PROGRAM.

- BULLET 1: THE RUNTIME SYSTEM KEEPS TRACK OF THE FACT THAT Heater_Task CANNOT PROCEED UNTIL ONE OF ITS ENTRIES HAS BEEN CALLED. A REASONABLE RUNTIME SYSTEM WILL NOT ALLOCATE THE CPU TO Heater_Task UNTIL THIS TAKES PLACE. IN THE PREVIOUS SCHEME, Heater_Task ALWAYS HAD WORK TO DO, POLLING THE ZONE TASKS.

- BULLET 2: IN THE PREVIOUS VERSION, Heater_Task REPEATEDLY QUERIED EACH ZONE TASK, THEN DID NOTHING WHEN IT FOUND THE VENT STATUS UNCHANGED. IN THIS VERSION, THE TASKS ONLY COMMUNICATE WHEN Heater_Task IS REQUIRED TO DO SOMETHING.

- BULLET 3: THE TASK IS SIMPLIFIED BECAUSE IT HAS LESS DETAIL TO DEAL WITH. THE SYSTEM IS SIMPLIFIED BECAUSE EACH TASK ONLY MAINTAINS DATA RELEVANT TO ITS OWN REAL-WORLD ACTIVITY. (THE NUMBER OF OPEN VENTS IS RELEVANT TO CONTROLLING THE HEATER, BUT THE STATUS OF EACH INDIVIDUAL VENT IS NOT.)

- BULLET 5: IN THE PREVIOUS VERSION, THE Zone_Task_Type LOOP MONITORING A ZONE'S TEMPERATURE WOULD HAVE HAD TO CONTAIN A STATEMENT LIKE

```
select
   accept Get_Vent_Status (Vent_Status:  out Vent_Status_Type) do
      Vent_Status:= Current_Status;
   end Get_Vent_Status;
else
   null;
end select;
```

  TO SERVE REQUESTS ARRIVING FROM Heater_Task AT UNPREDICTABLE INTERVALS.

ADVANTAGES

- Heater_Task DOES NOT EXPEND CPU TIME POLLING EACH ZONE FOR ITS CURRENT VENT STATUS, BUT "SLEEPS" UNTIL SOME VENT IS OPENED OR CLOSED.

- THE ZONE TASKS INTERACT WITH Heater_Task ONLY WHEN THERE IS NEWS TO REPORT.

- Heater_Task IS NOT CONCERNED WITH INDIVIDUAL ZONES.

  - WHICH ZONE IT IS INTERACTING WITH
  - THE CURRENT VENT STATUS OF EACH ZONE (ONLY TOTAL NUMBER OF OPEN VENTS MATTERS)

- Heater_Task HAS A SIMPLER CONTROL STRUCTURE

  - RUNTIME SYSTEM MANAGES INTERACTION WITH SEVERAL TASKS.
  - NO NEED TO TEST FOR CHANGE IN VENT STATUS (TASK ONLY AWAKENS AFTER STATUS HAS CHANGED).

- Zone_Task_Type HAS A SIMPLER CONTROL STRUCTURE.

  - NO NEED TO TEST WHETHER IT IS BEING POLLED.
  - NO EXTRA TESTS NEEDED TO CALL Heater_Task ENTRIES (ENTRIES ARE CALLED PRECISELY WHEN A VENT IS OPENED OR CLOSED).

INSTRUCTOR NOTES

ALLOW 60 MINUTES FOR THIS SECTION, NOT INCLUDING THE EXERCISE. TAKE A BREAK BEFORE THE EXERCISE, THEN ALLOW 75 MINUTES FOR SOLUTION AND DISCUSSION OF THE EXERCISE.

RECOMMEND THAT STUDENTS READ EXERCISE 4.2 IN THE REAL-TIME ADA WORKBOOK.

Section 12

MONITORS

INSTRUCTOR NOTES

THIS IS A BRIEF REVIEW OF MATERIAL COVERED IN SECTION 3, PLUS AN INTRODUCTION TO THE SUBJECT OF THIS SECTION.

MOVE ON QUICKLY.

THE SIMULTANEOUS UPDATE PROBLEM REVISITED

- WHEN TWO OR MORE TASKS ARE EXAMINING AND THEN MODIFYING THE SAME DATA, THEY CAN INTERFERE WITH EACH OTHER.

- THIS PROBLEM CAN BE SOLVED BY MUTUAL EXCLUSION.
  - PREVENT TWO TASKS FROM MANIPULATING THE DATA AT EXACTLY THE SAME TIME.

- MONITORS ARE Ada TASKS DESIGNED TO PROTECT DATA FROM SIMULTANEOUS ACCESS BY MORE THAN ONE OTHER TASK.

INSTRUCTOR NOTES

- BULLET 1:
  - ITEM 1: THE PULSES COULD BE RADAR BLIPS OR GEIGERS, FOR EXAMPLE.
  - ITEM 3: ASSUME THAT PULSES ARE QUEUED, SO THAT Wait_For_Pulse (N) DOES NOT WAIT IF A PULSE HAS ALREADY BEEN SENSED BY SENSOR N. SIMILARLY, IF 3 PULSES HAVE ARRIVED, Wait_For_Pulse CAN BE CALLED THREE TIMES WITHOUT WAITING. (BRING THIS UP ONLY IF ASKED).

- BULLET 2:
  - ITEM 2: ONE OF THESE TASKS IS ASSIGNED TO EACH SENSOR, TO INCREMENT Unreported_Pulses EACH TIME A PULSE IS RECEIVED BY THAT SENSOR.
  - ITEM 3: THE FUNCTION SAVES THE VALUE OF Unreported_Pulses, RESETS THAT VARIABLE TO ZERO (TO START COUNTING THE PULSES THAT WILL BE REPORTED BY THE NEXT FUNCTION CALL) AND RETURNS THE SAVED VALUE.

- BULLET 3:

  THE CLASS SHOULD RECOGNIZE BY NOW THAT THIS IS A SIMULTANEOUS UPDATE PROBLEM. WITH LUCK, THEY MAY BE ABLE TO POINT OUT SOME OF THE SPECIFIC THINGS THAT CAN GO WRONG:

  - IF ONE OF THE SENSOR TASKS INCREMENTS Unreported_Pulses WHILE New_Pulses IS BETWEEN ITS FIRST AND SECOND STATEMENTS, THE PULSE WILL NOT BE COUNTED.
  - IF TWO SENSOR TASKS BOTH EVALUATE THE RIGHT HAND SIDES OF THE ASSIGNMENT STATEMENTS, THEN BOTH PERFORM THE ASSIGNMENT, Unreported_Pulses WILL BE INCREMENTED BY 1 INSTEAD OF 2.
  - IF Unreported_Pulses HAS THE VALUE 99, A SENSOR TASK EVALUATES THE RIGHT HAND SIDE OF THE ASSIGNMENT, New_Pulses RESETS Unreported_Pulses TO 0, AND THE SENSOR TASK COMPLETES THE ASSIGNMENT, Unreported_Pulses WILL HAVE THE VALUE 100 INSTEAD OF 1.

AN EXAMPLE

- PROBLEM:
  - FIVE SENSORS, EACH RECEIVING PULSES
  - NEED A FUNCTION New_Pulses RETURNING THE TOTAL NUMBER OF PULSES RECEIVED ON ALL SENSORS SINCE THE LAST CALL ON THE FUNCTION.
  - INTERFACE: THE PROCEDURE CALL Wait_For_Pulse (N) CAUSES THE TASK CALLING IT TO WAIT UNTIL A PULSE IS DETECTED BY SENSOR N BEFORE IT PROCEEDS.
- AN INCORRECT SOLUTION:
  - A GLOBAL VARIABLE:

```
Unreported_Pulses : Natural := 0;
```

  - FIVE TASKS, EACH WITH ITS OWN VALUE FOR My_Sensor, EXECUTING THE FOLLOWING LOOP:

```
loop
   Wait_For_Pulse (My_Sensor);
   Unreported_Pulses := Unreported_Pulses + 1;
end loop;
```

  - THE New_Pulses FUNCTION BODY (EXECUTED BY A SIXTH TASK):

```
function New_Pulses return Natural is
   Result : Natural;
begin
   Result := Unreported_Pulses;
   Unreported_Pulses := 0;
   return Result;
end New_Pulses;
```

- WHY WON'T THIS WORK?

INSTRUCTOR NOTES

- BULLET 1:

  THE TASK Count_Manager HAS TWO ENTRIES, Increment_Count AND Obtain_And_Reset_Count. IT MAINTAINS A LOCAL VARIABLE NAMED Current_Count.

  A CALL ON Increment_Count CAUSES Current_Count TO BE INCREMENTED.

  A CALL ON Obtain_And_Reset_Count DELIVERS THE VALUE OF Current_Count TO THE CALLER. Current_Count IS THEN RESET TO ZERO.

  ALL Count_Manager EVER DOES IS SERVICE CALLS ON THESE ENTRIES.

- BULLET 2:

  THE SENSOR TASK NOW CALLS Count_Manager.Increment_Count INSTEAD OF EXECUTING
  Unreported_Pulses := Unreported_Pulses + 1.

- BULLET 3:

  New_Pulses NOW CALLS Count_Manager.Obtain_And_Reset (Result) INSTEAD OF EXECUTING

```
Result := Unreported_Pulses;
Unreported_Pulses := 0;
```

THE NEXT SLIDE ELABORATES ON WHY THIS SOLVES THE PROBLEM.

SOLUTION: A NEW TASK TO MANAGE THE SHARED DATA

- ADD THE FOLLOWING TASK:

```
task Count_Manager is
   entry Increment_Count;
   entry Obtain_And_Reset_Count (Old_Count : out Natural);
end Count_Manager;

task body Count_Manager is
   Current_Count : Natural := 0;
begin
   loop
      select
         accept Increment_Count;
         Current_Count := Current_Count + 1;
      or
         accept Obtain_And_Reset_Count (Old_Count : out Natural) do
            Old_Count := Current_Count;
         end Obtain_And_Reset_Count;
         Current_Count := 0;
      end select;
   end loop;
end Count_Manager;
```

- CHANGE THE SENSOR TASK LOOP TO:

```
loop
   Wait_For_Pulse (My_Sensor);
   Count_Manager.Increment_Count;
end loop;
```

- CHANGE New_Pulses TO:

```
functional New_Pulses return Natural is
   Result : Natural;
begin
   Count_Manager.Obtain_And_Reset_Count (Result);
   return Result;
end New_Pulses;
```

THIS SOLUTION WORKS BECAUSE IT ENFORCES MUTUAL EXCLUSION AMONG THE FIVE SENSOR TASKS AND THE TASK EXECUTING New_Pulses. DIFFERENT USES OF THE SHARED COUNT CANNOT BE IN PROGRESS AT THE SAME TIME.

THIS IS ACHIEVED BY HIDING THE SHARED COUNT IN THE Count_Manager TASK BODY, SO OTHER TASKS ARE FORCED TO CALL UPON Count_Manager TO MANIPULATE THE COUNT FOR THEM.

Count_Manager DOES ONE THING AT A TIME, AND COMPLETELY SERVICES ONE REQUEST TO MANIPULATE THE COUNT BEFORE ACCEPTING ANOTHER REQUEST. (EVEN THOUGH SEVERAL TASKS MAY BE CALLING Count_Manager's ENTRIES SIMULTANEOUSLY, Count_Manager ACCEPTS ENTRY CALLS ONE AT A TIME, ACCEPTING ONE CALL EACH TIME IT REACHES AN ACCEPT STATEMENT).

- BULLET 3:

  THE NOTION OF AN INDIVISIBLE OPERATION IS AN IMPORTANT ONE. THERE IS NO HARM IN COMPLETE INCREMENT OPERATIONS AND COMPLETE REPORT/RESET OPERATIONS BEING INTERLEAVED. IN FACT, THAT IS PRECISELY WHAT WE WANT AND EXPECT. IT IS THE POSSIBILITY OF PARTS OF THESE OPERATIONS BEING INTERLEAVED THAT POSES A DANGER. AN INDIVISIBLE OPERATION ON AN OBJECT IS ONE THAT MUST BE PERFORMED AS A UNIT -- WITH NO INTERVENING OPERATIONS ON THE SAME OBJECT BY ANOTHER TASK -- TO GUARANTEE ITS CORRECTNESS. ONLY IF THE ASSIGNMENT Current_Count := Current_Count + 1 IS EXECUTED INDIVISIBLY -- WITH Current_Count UNALTERED BETWEEN THE EVALUATION OF THE RIGHT HAND SIDE AND THE UPDATING OF THE LEFT HAND SIDE -- CAN WE BE SURE THAT IT INCREASES Current_Count BY 1.

Count_Manager ENFORCES MUTUAL EXCLUSION

- THE VARIABLE Current_Count IS ONLY ACCESSIBLE IN THE Count_Manager TASK BODY.

```
task body Count_Manager is
   Current_Count : Natural := 0;
begin
   loop
      select
         accept Increment_Count;
         Current_Count := Current_Count + 1;
      or
         accept Obtain_And_Reset_Count (Old_Count : out Natural) do
            Old_Count := Current_Count;
         end Obtain_And_Reset_Count;
         Current_Count := 0;
      end select;
   end loop;
end Count_Manager;
```

- AT ANY MOMENT, THE Count_Manager TASK CAN ONLY BE AT ONE POINT IN ITS SEQUENCE OF STATEMENTS.

- THE Count_Manager TASK BODY COMPLETES ONE OPERATION ON THE VARIABLE BEFORE BEGINNING ANOTHER.

  - IN THIS CONTEXT "OPERATION" MEANS SOMETHING THAT SHOULD BE DONE INDIVISIBLY, WITHOUT ANY OTHER USES OF THE VARIABLES INTERLEAVED.

  - EXAMPLE: BETWEEN THE TIME Current_Count IS ASSIGNED TO Old_Count AND THE TIME IT IS RESET TO ZERO, Count_Manager DOES NOT ALLOW Current_Count TO BE MANIPULATED IN ANY OTHER WAY.

INSTRUCTOR NOTES

THE TERM MONITOR WAS ORIGINALLY INTRODUCED BY C.A.R. HOARE (COMMUNICATIONS OF THE ACM, OCTOBER 1974, PP. 549-557), WHO USED IT TO DESCRIBE A SPECIFIC, SOMEWHAT MORE INTRICATE MECHANISM FOR PROVIDING RESTRICTED, MUTUALLY EXCLUSIVE OPERATIONS ON SOME DATA STRUCTURE. PER BRINCH-HANSEN (OPERATING SYSTEM PRINCIPLES, PRENTICE-HALL, 1973, P. 336) DEFINES A MONITOR MORE GENERALLY AS "A COMMON DATA STRUCTURE AND A SET OF MEANINGFUL OPERATIONS ON IT THAT EXCLUDE ONE ANOTHER IN TIME AND CONTROL THE SYNCHRONIZATION OF CONCURRENT PROCESSES." THAT IS THE DEFINITION USED IN THIS MODULE. ("PROCESS SYNCHRONIZATION" SIMPLY REFERS TO THE FACT THAT ONE PROCESS MAY BE FORCED TO WAIT WHILE THE MONITOR SERVICES ANOTHER TASK.)

IN THE CASE OF Count_Manager, THE DATA STRUCTURE IS A SIMPLE VARIABLE OF SUBTYPE Natural. LATER IN THE SECTION, WE WILL SEE A MORE COMPLEX DATA STRUCTURE.

DEFINITION OF A MONITOR

- THE Count_Manager TASK IS AN EXAMPLE OF A MONITOR.

- GENERAL DEFINITION:

  IN Ada, A MONITOR IS A TASK DESIGNED TO RESTRICT ACCESS TO A DATA STRUCTURE.

  - THE DATA STRUCTURE IS DECLARED IN THE TASK BODY.

  - THE DATA STRUCTURE CAN ONLY BE MANIPULATED THROUGH THE ABSTRACT OPERATIONS IMPLEMENTED AS ENTRIES TO THE MONITOR.

  - THE DATA STRUCTURE CANNOT BE SIMULTANEOUSLY MANIPULATED BY MORE THAN ONE TASK.

INSTRUCTOR NOTES

- BULLET 1:

  WHEN TASKS WERE FIRST DESCRIBED AS DATA OBJECTS, WE CONCEDED THAT THE IDEA MIGHT SEEM UNINTUITIVE AT FIRST, BUT PROMISED THAT IT WOULD SEEM MORE NATURAL LATER ON.

  WE NOW VENTURE TO FULFILL THAT PROMISE. IT IS NATURAL TO THINK OF A MONITOR AS A SPECIAL KIND OF DATA OBJECT THAT CAN BE SAFELY OPERATED ON BY CONCURRENT TASKS.

- BULLET 2:

  THIS IS THE SAME NOTATION USED WHEN PACKAGES ARE VIEWED AS OBJECTS. (IN THAT CASE, THE OBJECT NAME IS REALLY A PACKAGE NAME AND THE OPERATION NAME IS REALLY THE NAME OF ONE OF THE SUBPROGRAMS PROVIDED BY THE PACKAGE. FOR EXAMPLE, A PACKAGE NAMED Message_Stack MIGHT PROVIDE Push AND Pop OPERATIONS THAT OPERATE ON VARIABLES DECLARED INSIDE THE PACKAGE. THESE OPERATIONS ARE INVOKED BY CALLS LIKE Message_Stack.Push (X)).

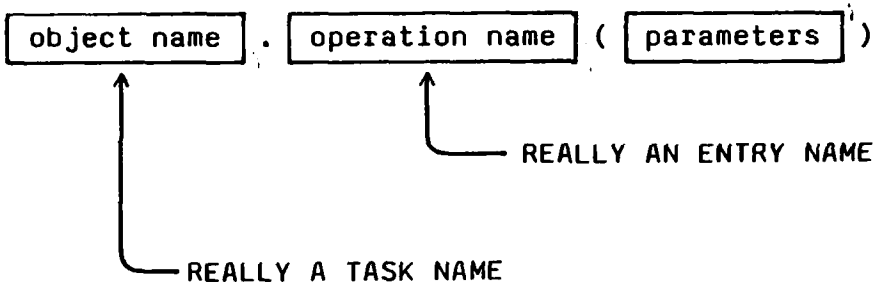
- BULLET 3:

  Count_Manager AND Read_And_Obtain_Count HAVE BEEN RENAMED Pulse_Count AND Read_And_Reset IN ACCORDANCE WITH THE VIEW THAT THE MONITOR IS A DATA OBJECT AND ITS ENTRIES ARE OPERATIONS.

MONITORS AS DATA OBJECTS

- OFTEN IT IS CONVENIENT TO THINK OF A MONITOR NOT AS A PROGRAM UNIT PROTECTING A DATA OBJECT, BUT AS A DATA OBJECT ITSELF.

- "OPERATIONS" ON THE OBJECT ARE WRITTEN:



- EXAMPLE:

```
task Pulse_Count is
   entry Increment;
   entry Read_And_Reset (Old_Value : out Natural);
end Pulse_Count;

--  OPERATIONS ON THE OBJECT Pulse_Count are:
--     Pulse_Count.Increment;
--     Pulse_Count.Read_And_Reset (Old_Value => X);
```

INSTRUCTOR NOTES

THE EXAMPLE SHOWS THE SINGLE TASK DECLARATION FOR Count_Manager TRANSFORMED INTO A TASK TYPE DECLARATION.

Count_Manager IS THEN DECLARED AS ONE OBJECT IN THIS TYPE, AND CAN BE USED AS BEFORE.

THE ARRAY DECLARATION SHOWS HOW WE COULD MAINTAIN SEVERAL SHARED COUNTS BY USING THE MONITOR TYPE AS THE COMPONENT TYPE OF AN ARRAY. IF EACH PULSE BELONGED TO ONE OF SEVERAL "FREQUENCY BANDS" AND Wait_For_Pulse HAD A SECOND PARAMETER RETURNING THE FREQUENCY BAND OF THE DETECTED PULSE, A SENSOR TASK COULD CALL THE Increment ENTRY OF THE APPROPRIATE COMPONENT OF Pulse_Count_List TO MAINTAIN SEPARATE COUNTS FOR EACH FREQUENCY BAND:

```
loop
   Wait_For_Pulse (My_Sensor, Frequency_Band);
   Pulse_Count_List (Frequency_Band).Increment;
end loop;
```

(ASSURE THAT Frequency_Band_Type IS SOME DISCRETE TYPE AND THAT Frequency_Band BELONGS TO THIS TYPE.)

ASK THE CLASS TO COME UP WITH THE SECOND STATEMENT IN THIS LOOP.

TASK TYPES FOR MONITORS

- IF WE THINK OF MONITORS AS DATA OBJECTS, IT IS NATURAL TO DECLARE TYPES WHOSE VALUES ARE MONITORS.

- APPROPRIATE WHEN SEVERAL OBJECTS OF THE SAME TYPE ARE TO BE
  - MANIPULATED WITH THE SAME ABSTRACT OPERATIONS
  - PROTECTED FROM SIMULTANEOUS ACCESS BY MORE THAN ONE TASK

- EXAMPLE:

```
task type Shared_Accumulator_Type is
   entry Increment;
   entry Read_And_Reset (Old_Value : out Natural);
end Shared_Accumulator_Type;

Pulse_Count_List : array (Frequency_Band_Type) of Shared_Accumulator_Type;
```

- SUPPOSE Wait_For_Pulse WERE REDEFINED AS FOLLOWS:

```
procedure Wait_For_Pulse
   (Sensor : in Sensor_ID_Type; Band : out Frequency_Band_Type);
   -- WAIT FOR A PULSE ON THE SPECIFIED SENSOR AND SET BAND
   -- TO THE FREQUENCY BAND OF THE DETECTED PULSE.
```

COMPLETE THE FOLLOWING SENSOR TASK LOOP TO USE Pulse_Count_List TO MAINTAIN SEPARATE COUNTS FOR EACH FREQUENCY BAND:

```
loop
   Wait_For_Pulse (My_Sensor, Frequency_Band);
   [ ? ]
end loop;
```

INSTRUCTOR NOTES

- BULLET 1:

  ANY SUBPROGRAM, BLOCK, OR TASK DECLARING AN OBJECT IN A MONITOR TYPE BECOMES THE MASTER OF THAT OBJECT. THIS MEANS THAT THE SUBPROGRAM, BLOCK, OR TASK CANNOT TERMINATE UNTIL THE MONITOR TERMINATES.

  IF THE MONITOR DOES NOT CONTAIN A TERMINATE ALTERNATIVE, IT WILL NEVER TERMINATE. IF IT DOES CONTAIN A TERMINATE ALTERNATIVE, IT CAN TERMINATE WHEN (FOR EXAMPLE) ITS MASTER IS COMPLETE AND ALL TASKS DEPENDENT ON THE MASTER (E.G. OTHER MONITOR OBJECTS) ARE WAITING AT SELECT STATEMENTS WITH TERMINATE ALTERNATIVES. (REVIEW THE DISTINCTION BETWEEN COMPLETED AND TERMINATED.)

- BULLET 2:

  ORDINARILY, WHEN THE WRITER OF A SUBPROGRAM DECLARES A DATA OBJECT, HE NEED NOT BE CONCERNED THAT THE DECLARATION WILL PREVENT THE SUBPROGRAM FROM TERMINATING.

- BULLET 3:

  THIS IS JUST THE Count_Manager TASK BODY WITH THE NAME CHANGED AND A TERMINATE ALTERNATIVE ADDED.

MONITOR TYPES AND terminate ALTERNATIVES

DESIGN HINT

THE TASK BODY FOR A MONITOR TYPE SHOULD INCLUDE A terminate ALTERNATIVE.

- THIS ALLOWS THE MONITOR TO TERMINATE WHEN ITS MASTER IS COMPLETE.
  - FREE RESOURCES
  - ALLOWS MASTER TO TERMINATE
- MONITORS THEN BEHAVE MORE LIKE ORDINARY DATA OBJECTS.
- EXAMPLE:

```
task body Shared_Accumulator_Type is
   Current_Count : Natural := 0;
begin -- Shared_Accumulator_Type
   loop
      select
         accept Increment;
         Current_Count := Current_Count + 1;
      or
         accept Read_And_Reset (Old_Value : out Natural) do
            Old_Value := Current_Count;
         end Read_And_Reset;
         Current_Count := 0;
      or
         terminate;
      end select;
   end loop;
end Shared_Accumulator_Type;
```

INSTRUCTOR NOTES

WE ARE NOT PRESENTING Ada SYNTAX RULES HERE, BUT ONE PARTICULAR FORM THAT A TASK TYPE DECLARATION AND TASK TYPE BODY CAN TAKE.

(THE GENERAL SYNTAX WAS GIVEN IN SECTION 5.)

POINT OUT THE FOLLOWING:

- THE WORD TYPE IN THE TASK TYPE DECLARATION.
- ENTRY DECLARATIONS FOR SHARED DATA.
- DECLARATIONS FOR LOCAL DATA INSIDE THE TASK BODY.
- THE BASIC STRUCTURE OF A SELECT STATEMENT INSIDE A NESTED LOOP.
- ACCEPT ALTERNATIVES FOR EACH ENTRY
  (WE HAVE NOT YET SHOWN A USE FOR GUARDS, BUT WILL AT THE END OF THE SECTION).
- THE TERMINATE ALTERNATIVE.

OCCASIONALLY MONITORS VARY FROM THIS TYPICAL FORM BY INCLUDING ADDITIONAL STATEMENTS BEFORE THE LOOP (FOR INITIALIZATION) OR WITHIN THE LOOP BEFORE OR AFTER THE SELECT STATEMENT (E.G. TO EVALUATE COMMON SUBEXPRESSIONS USED IN MORE THAN ONE GUARD).

TYPICAL FORM OF A MONITOR

```
task type [identifier] is
{ [entry declaration for an abstract
   operation on shared data] }
end [identifier];


task body [identifier] is
   [declarations, including data to be
   protected from simultaneous update]
begin -- [identifier]
   loop
      select
      { [when [condition] =>]
           [accept alternative for one
           of the abstract operations]
      or }
           terminate;
      end select;
   end loop;
end [identifier];
```

INSTRUCTOR NOTES

- BULLET 1:

  SUCH A TYPE COULD BE USED IN A REAL-TIME SYSTEM TO SCHEDULE EVENTS FOR SPECIFIC TIMES AND DELAY UNTIL THE TIME OF THE NEXT SCHEDULED EVENT OR THE NEXT ENTRY CALL, WHICHEVER COMES FIRST.

- BULLET 2:

  THE DEFINITIONS OF Activity_Type DOES NOT CONCERN US HERE. (IT IS SOME DESCRIPTION OF THE ACTIVITY TO BE PERFORMED AT SOME SPECIFIED TIME.)

  THIS PACKAGE DECLARATION INCLUDES THE TASK TYPE DECLARATION FOR THE MONITOR TYPE Agenda_Type THAT THE PACKAGE PROVIDES. THE Agenda_Type TASK BODY LOGICALLY GOES IN THE Agenda_Package BODY, SHOWN ON THE NEXT SLIDE.

THE POINTS OF THIS EXAMPLE ARE (1) THAT MONITORS CAN BE USED NOT ONLY FOR SIMPLE DATA LIKE INTEGER VARIABLES, BUT ALSO FOR MORE COMPLEX DATA STRUCTURES: (2) THAT WHEN THIS HAPPENS, THE MONITOR IS IMPORTANT NOT ONLY AS A SYNCHRONIZATION TUOL (TO PROVIDE MUTUAL EXCLUSION), BUT ALSO AS A DATA ABSTRACTION TOOL (TO HIDE IMPLEMENTATION DETAILS).

A MORE INTRICATE EXAMPLE

- MONITOR TYPE FOR AGENDAS. OPERATIONS ARE:
  - SCHEDULING A SPECIFIED ACTIVITY FOR A SPECIFIED TIME
  - EXAMINING THE NEXT SCHEDULED ACTIVITY (WITHOUT REMOVING IT FROM THE SCHEDULE)
  - REMOVING THE NEXT SCHEDULED ACTIVITY FROM THE SCHEDULE.

- A PACKAGE PROVIDES A TYPE FOR NAMING ACTIVITIES AND A MONITOR TYPE FOR SCHEDULES:

```
with Calendar; use Calendar;

package Agenda_Package is

   type Activity_Type is ... ;

   task type Agenda_Type is
      entry Schedule_Activity (Activity : in Activity_Type;
                               Starting_Time : in Time);
      entry Identify_Next_Activity (Activity : out Activity_Type;
                                    Starting_Time : out Time);
      entry Advance_To_Next_Activity;
   end Agenda_Type;

end Agenda_Package;
```

INSTRUCTOR NOTES

THIS SLIDE SIMPLY SETS UP THE CONTEXT FOR THE Agenda_Type TASK BODY, GIVEN AS A SUBUNIT ON THE NEXT SLIDE.

List_Type IMPLEMENTS A SINGLY-LINKED LIST, EACH CELL OF WHICH HAS AN Activity_Part, A Time_Part, AND A LINK TO THE NEXT CELL.

DEALLOCATE IS AN INSTANCE OF THE PREDEFINED LIBRARY GENERIC PROCEDURE Unchecked Deallocation, FOR RECYCLING LIST CELLS.

MOVE ON QUICKLY.

Agenda_Package BODY

- THE DATA STRUCTURE TO BE PROTECTED BY THE MONITOR IS A LINKED LIST SORTED IN ORDER OF ASCENDING Time_Part VALUES.

```
with Unchecked_Deallocation;

package body Agenda_Package is

   type List_Cell_Type;
   type List_Type is access List_Cell_Type;
   type List_Cell_Type is
      record
         Activity_Part : Activity_Type;
         Time_Part     : Time;
         Link_Part     : List_Type;
      end record;

   procedure Deallocate is
      new Unchecked_Deallocation (List_Cell_Type, List_Type);

   task body Agenda_Type is separate;

end Agenda_Package;
```

INSTRUCTOR NOTES

ATTENTION, INSTRUCTOR! THIS IS NOT A COURSE IN DATA STRUCTURES. DO NOT WALKTHROUGH THE LIST MANIPULATION STEP-BY-STEP.

THE IMPRESSION TO CONVEY IS THAT THIS MONITOR PERFORMS THE KIND OF COMPLEX DATA STRUCTURE MANIPULATION THAT STUDENTS SAW IN MODULE L305.

STATE THAT

1. THE FIRST ACCEPT ALTERNATIVE INSERTS A NEW TIME AND ACTIVITY AT THE APPROPRIATE POSITION IN THE LIST;

2. THE SECOND ACCEPT ALTERNATIVE RETURNS THE TIME AND ACTIVITY AT THE FRONT OF A NONEMPTY LIST; AND

3. THE THIRD ACCEPT ALTERNATIVE REMOVES THE FIRST ITEM ON THE LIST AND RECYCLES ITS STORAGE;

BUT DO NOT EXPLAIN WHY THIS IS SO. ASK THE STUDENTS TO TAKE THIS ON FAITH AND STUDY THE EXAMPLE LATER IF THEY ARE SO INCLINED.

POINT OUT THE GUARDS ON THE SECOND AND THIRD ALTERNATIVES, TO ENSURE THAT OPERATIONS ON THE FIRST ITEM IN A LIST ARE NOT ATTEMPTED FOR EMPTY LISTS. FROM AN EXTERNAL VIEW, THIS MEANS THAT CALLS ON Identify_Next_Activity AND Advance_To_Next_Activity MUST WAIT UNTIL SOME ACTIVITY IS SCHEDULED.

Agenda_Type TASK BODY

```
separate (Agenda_Package)
task body Agenda_Type is
   Dummy_Cell        : List_Type := new List_Cell_Type;
   First_Item        : List_Type renames Dummy_Cell.Link_Part;
   Next, Old_Cell    : List_Type;
   New_Activity      : Activity_Type;
   New_Time          : Time;
begin -- Agenda_Type
   loop
      select
         accept Schedule_Activity (Activity : in Activity_Type; Starting_Time : in Time) do
            New_Activity := Activity;
            New_Time := Starting_Time;
         end Schedule_Activity;
         Next := Dummy_Cell;
         while Next.Link_Part /= null and then New_Time >= Next.Link_Part.Time_Part loop
            Next := Next.Link_Part;
         end loop;
         Next.Link_Part := new List_Cell_Type (New_Activity, New_Time, Next.Link_Part);
      or
         when First_Item /= null =>
            accept Identify_Next_Activity
               (Activity : out Activity_Type; Starting_Time : out Time) do
               Activity := First_Item.Activity_Part;
               Starting_Time := First_Item.Time_Part;
            end Identify_Next_Activity;
      or
         when First_Item /= null =>
            accept Advance_To_Next_Activity;
            Old_Cell := First_Item;
            First_Item := First_Item.Link_Part;
            Deallocate (Old_Cell);
      or
         terminate;
      end select;
   end loop;
end Agenda_Type;
```

INSTRUCTOR NOTES

- BULLET 2:

  THIS POINT IS CRUCIAL. MAKE SURE STUDENTS UNDERSTAND IT.

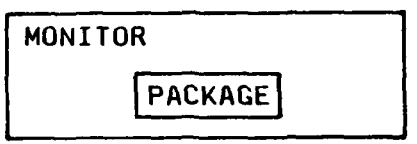
  - ITEM 2: THE LOGIC OF THE SUBPROGRAM BODIES MAY BE BASED ON THE ASSUMPTION THAT ONE SUBPROGRAM COMPLETES BEFORE ANOTHER STARTS.

- BULLET 3:

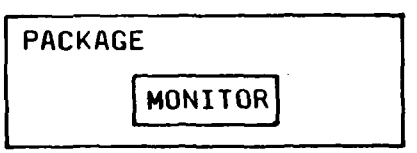
  - ITEM 1: THIS APPROACH, DETAILED ON THE NEXT TWO SLIDES, INVOLVES PUTTING A PACKAGE INSIDE A MONITOR.
  - ITEM 2: THIS APPROACH, DETAILED IN THE TWO SUBSEQUENT SLIDES, INVOLVES PUTTING A MONITOR INSIDE A PACKAGE BODY.

MONITORS AND PACKAGES

- BOTH CAN BE USED TO ENCAPSULATED DATA STRUCTURES
  - PROVIDE A DATA ABSTRACTION.
  - PREVENT THE DATA FROM BEING MANIPULATED EXCEPT BY PRESCRIBED OPERATIONS.
  - HIDE IMPLEMENTATION DETAILS.
- UNLIKE MONITORS, PACKAGES DO NOT PROTECT AGAINST SIMULTANEOUS UPDATE!
  - SINCE Ada SUBPROGRAMS ARE REENTRANT, TWO TASKS MAY BE SIMULTANEOUSLY EXECUTING SUBPROGRAMS PROVIDED BY THE PACKAGE.
  - IF BOTH TASKS HAVE PASSED THE SAME OBJECT (OR POINTERS TO THE SAME OBJECT) AS PARAMETERS, DIFFERENT OPERATIONS ON THE OBJECT MAY BE INTERLEAVED.
    - o THIS MAY INVALIDATE THE OPERATIONS.
  - IF THE SUBPROGRAMS MANIPULATE A VARIABLE DECLARED IN THE PACKAGE, THAT VARIABLE IS SUBJECT TO SIMULTANEOUS UPDATE.
- TWO APPROACHES TO MAKING PACKAGES SHAREABLE
  - ENCLOSE THE PACKAGE IN A MONITOR TO PROTECT IT FROM SIMULTANEOUS USE BY MORE THAN ONE TASK.



  - USE A MONITOR TO IMPLEMENT THE PACKAGE.



INSTRUCTOR NOTES

THIS IS THE FIRST OF TWO SLIDES ILLUSTRATING SLIDE 12-13'S FIRST APPROACH TO MAKING PACKAGES SHAREABLE.

- BULLET 1:

  VARIABLES IMPLEMENTING SOME VERSION OF A PRIORITY QUEUE ARE DECLARED INSIDE THE PACKAGE BODY. THERE ARE NO PRIORITY QUEUE PARAMETERS; THE PACKAGE PROVIDES A SINGLE (HIDDEN) PRIORITY QUEUE OBJECT MANIPULATED BY ALL OF THE PACKAGE'S ROUTINES.

  Empty_Queue_Error IS RAISED IF Examine_Front OR Dequeue_Activity IS CALLED WHEN Queue_Empty IS TRUE.

- BULLET 3:

  THIS IS ILLUSTRATED ON THE NEXT SLIDE.

ENCLOSING THE PACKAGE IN A MONITOR

- SUPPOSE THE FOLLOWING PACKAGE (HIDING DECLARED VARIABLES AND DESIGNED FOR USE BY A SINGLE TASK) WERE AVAILABLE IN A LIBRARY:

```
package Activity_Priority_Queue is

   type Activity_Type is ... ;
   procedure Enqueue_Activity (Activity : in Activity_Type; Starting_Time : in Time);
   function Queue_Empty return Boolean;
   procedure Examine_Front (Activity : out Activity_Type; Starting_Time : out Time);
   procedure Dequeue_Activity;
   Empty_Queue_Error: exception;

end Activity_Priority_Queue;
```

- THIS PACKAGE COULD BE PROTECTED FROM SIMULTANEOUS UPDATE BY ENCLOSING IT IN A MONITOR WITH THE INTERFACE DESCRIBED EARLIER:

```
task type Agenda_Type is
   entry Schedule_Activity (Activity : in Activity_Type; Starting_Time : in Time);
   entry Identify_Next_Activity (Activity : out Activity_Type; Starting_Time : out Time);
   entry Advance_To_Next_Activity;
end Agenda_Type;
```

- THE MONITOR HANDLES EACH ENTRY CALL SIMPLY BY CALLING ONE OF THE PACKAGE'S SUBPROGRAMS.

  - THE MONITOR HAS PROVIDED MUTUAL EXCLUSION.

- THIS APPROACH IS APPROPRIATE IF AN EXISTING PACKAGE DESIGNED FOR USE BY ONE TASK IS TO BE ADAPTED FOR USE BY SEVERAL TASKS.

INSTRUCTOR NOTES

THIS IS THE SECOND OF TWO SLIDES ILLUSTRATING SLIDE 12-13'S FIRST APPROACH TO MAKING PACKAGES SHAREABLE.

EACH ENTRY CORRESPONDS TO ONE OF THE PACKAGE'S OPERATIONS, AND THE ACCEPT ALTERNATIVE FOR EACH ENTRY DOES LITTLE MORE THAN CALL THE CORRESPONDING SUBPROGRAM OF THE PACKAGE.

IN THE CASE OF Schedule_Activity THE ACCEPT STATEMENT ONLY COPIES PARAMETERS. Enqueue_Activity, CONTAINING A POTENTIALLY EXPENSIVE LOOP TO INSERT AN ITEM AT THE RIGHT PLACE IN A LIST, IS NOT CALLED UNTIL THE RENDEZVOUS IS OVER. THIS ALLOWS THE CALLING TASK TO GO ABOUT ITS BUSINESS AS SOON AS POSSIBLE.

THE GUARDS ON THE SECOND AND THIRD ALTERNATIVES PREVENT Examine_Front AND Dequeue_Activity FROM BEING CALLED IN THE CASE WHERE THEY WOULD RAISE AN EXCEPTION. SINCE EACH GUARD IS EVALUATED AT THE BEGINNING OF A SELECT STATEMENT, THE ASSIGNMENT AT THE TOP OF THE LOOP CAUSES Queue_Empty TO BE CALLED ONCE PER ITERATION INSTEAD OF TWICE.

IMPLEMENTATION OF THE MONITOR IN TERMS OF THE PACKAGE

```
task body Agenda_Type is
   Something_Scheduled : Boolean;
   New_Activity        : Activity_Type;
   New_Time            : Time;
begin  -- Agenda_Type
   loop
      Something_Scheduled := not Activity_Priority_Queue.Queue_Empty;
      select
         accept Schedule_Activity (Activity : in Activity_Type; Starting_Time : in Time) do
            New_Activity := Activity;
            New_Time := Starting_Time;
         end Schedule_Activity;
         Activity_Priority_Queue.Enqueue_Activity (New_Activity, New_Time);
      or
         when Something_Scheduled =>
            accept Identify_Next_Activity
               (Activity : out Activity_Type; Starting_Time : out Time) do
               Activity_Priority_Queue.Examine_Front (Activity, Starting_Time);
            end Identify_Next_Activity;
      or
         when Something_Scheduled =>
            accept Advance_To_Next_Activity;
            Activity_Priority_Queue.Dequeue_Activity;
      or
         terminate;
      end select;
   end loop;
end Agenda_Type;
```

INSTRUCTOR NOTES

THIS IS THE FIRST OF TWO SLIDES ILLUSTRATING SLIDE 12-13'S SECOND APPROACH TO MAKING PACKAGES SHAREABLE.

IN THIS CASE WE ARE WRITING A PACKAGE PROVIDING A TYPE FOR AGENDAS AND THREE PROCEDURES FOR THE MANIPULATION OF AGENDAS. IT SHOULD BE POSSIBLE FOR SEVERAL TASKS TO USE THE PACKAGE CONCURRENTLY.

- BULLET 1:

  THE TYPE MUST BE DECLARED LIMITED PRIVATE IN THE VISIBLE PART IF IT IS DECLARED AS A TASK TYPE IN THE PRIVATE PART.

  THIS EXAMPLE REINFORCES THE NOTION OF TASK-AS-OBJECT. THE USER OF THE PACKAGE NEED NOT EVEN KNOW THAT Agenda_Type OBJECTS ARE REPRESENTED AS TASK OBJECTS.

- BULLET 2:
  - ITEM 2: WE SAW AN EXAMPLE OF THIS BACK ON SLIDE 12-3. THE FUNCTION New_Pulses SIMPLY CALLED AN ENTRY AND RETURNED THE VALUE TO WHICH THE ENTRY SET ITS PARAMETER.

THE NEXT SLIDE GIVES THE PACKAGE BODY.

USING A MONITOR TO IMPLEMENT A PACKAGE

```
with Calendar; use Calendar;
package Shared_Agenda_Package is
   type Activity_Type is ... ;
   type Agenda_Type is limited private;
   procedure Add_Activity
      (Agenda : in out Agenda_Type; Activity : in Activity_Type; Starting_Time : in Time);
   procedure Examine_Next_Activity
      (Agenda : in Agenda_Type; Activity : out Activity_Type; Starting_Time : out Time);
   procedure Remove_Activity (Agenda : in out Agenda_Type);
private
   task type Agenda_Type is
      entry Schedule_Activity (Activity : in Activity_Type; Starting_Time : in Time);
      entry Identify_Next_Activity (Activity : out Activity_Type; Starting_Time : out Time);
      entry Advance_To_Next_Activity;
   end Agenda_Type;
end Shared_Agenda_Package;
```

- A LIMITED PRIVATE TYPE DECLARED IN THE VISIBLE PART OF A PACKAGE CAN BE DECLARED IN THE PRIVATE PART WITH A TASK TYPE DECLARATION.
- THIS APPROACH IS APPROPRIATE IF A DATA ABSTRACTION WAS DESIGNED AS A MONITOR IN THE FIRST PLACE.
  - IT PROVIDES THE SAME KIND OF EXTERNAL INTERFACE AS FOR NONSHARED DATA ABSTRACTIONS.
  - WHERE APPROPRIATE, AN ENTRY CALL WITH ONE out PARAMETER CAN BE USED TO IMPLEMENT A FUNCTION.

INSTRUCTOR NOTES

THIS IS THE SECOND OF TWO SLIDES ILLUSTRATING SLIDE 12-13'S SECOND APPROACH TO MAKING PACKAGES SHAREABLE.

EACH OF THE PACKAGE'S SUBPROGRAMS CORRESPONDS TO ONE OF Agenda_Type's ENTRY.

EACH SUBPROGRAM SIMPLY CALLS THE CORRESPONDING ENTRY OF ITS Agenda_Type PARAMETER, USING THE OTHER SUBPROGRAM PARAMETERS AS ENTRY CALL PARAMETERS.

IMPLEMENTATION OF THE PACKAGE IN TERMS OF THE MONITOR

```
package body Shared_Agenda_Package is

   task body Agenda_Type is separate;  -- Not shown here; same as before.

   procedure Add_Activity
      (Agenda : in out Agenda_Type; Activity : in Activity_Type; Starting_Time : in Time) is
   begin
      Agenda.Schedule_Activity (Activity, Starting_Time);
   end Add_Activity;

   procedure Examine_Next_Activity is
      (Agenda : in Agenda_Type; Activity : out Activity_Type; Starting_Time : out Time) is
   begin
      Agenda.Identify_Next_Activity (Activity, Starting_Time);
   end Examine_Next_Activity;

   procedure Remove_Activity (Agenda : in out Agenda_Type) is
   begin
      Agenda.Advance_To_Next_Activity;
   end Remove_Activity;

end Shared_Agenda_Package;
```

- TASK TYPE PARAMETERS CAN BE USED INSIDE A SUBPROGRAM BODY AS THE TASK NAMES IN ENTRY CALLS.
  - OTHER PARAMETERS TO THE SUBPROGRAM CAN BE USED AS ACTUAL PARAMETERS IN THE ENTRY CALL.

INSTRUCTOR NOTES

- BULLET 1: ANY EXPERIENCED PROGRAMMER SHOULD HAVE AT LEAST AN INTUITIVE UNDERSTANDING OF THIS NOTION.

- BULLET 2: THE OPERATIONS DESCRIBED IN ITEM 2 ARE THE PREFERRED WAY OF DETERMINING WHETHER THE PRECONDITIONS HOLD. EXCEPTIONS ARE A WAY FOR THE PACKAGE TO PROTECT ITSELF FROM A CALLER THAT FAILS TO USE THOSE OPERATIONS. THE REASON FOR THE PREFERENCE IS THAT THE FLOW OF CONTROL IN

```
if not Stack_Object_Package.Stack_Is_Empty then
   Stack_Object_Package.Pop (X);
end if;
```

  FOR EXAMPLE, IS CLEARER THAN

```
begin
   Stack_Object_Package.Pop (X);
exception
   when Pop_Error => null;
end;
```

- BULLET 3:

  - ITEM 1: THIS IS JUST THE SIMULTANEOUS UPDATE PROBLEM ALL OVER AGAIN. THE MONITOR SHOULD COMBINE TESTING FOR AN OPERATION'S PRECONDITION AND CONDITIONALLY PERFORMING THE OPERATION INTO ONE INDIVISIBLE OPERATION IN THE SENSE OF SLIDE 12-4.

  - ITEM 3: THIS IS ILLUSTRATED ON THE NEXT SLIDE.

  - ITEM 4: FOR EXAMPLE, A CALL ON A Pop ENTRY MIGHT WAIT UNTIL ANOTHER TASK CALLS A Push ENTRY. IF WAITING IS UNACCEPTABLE, A CONDITIONAL ENTRY CALL CAN BE ISSUED.

PRECONDITIONS OF ABSTRACT OPERATIONS

- SOME OPERATIONS ON DATA ABSTRACTIONS CAN ONLY BE PERFORMED IF CERTAIN PRECONDITIONS HOLD BEFOREHAND.
  - PRECONDITION FOR PUSHING ONTO A STACK: THE STACK IS NOT FULL.
  - PRECONDITION FOR POPPING OFF A STACK: THE STACK IS NOT EMPTY.
- PACKAGES OFTEN PROVIDE
  - EXCEPTIONS RAISED WHEN AN OPERATION IS INVOKED AND ITS PRECONDITION DOES NOT HOLD.
  - OPERATIONS TO DETERMINE WHETHER OTHER OPERATIONS' PRECONDITIONS HOLD.

```
package Stack_Object_Package is
   procedure Push (Item : in Integer);
   procedure Pop  (Item : out Integer);
   function Stack_Is_Full return Boolean;
   function Stack_Is_Empty return Boolean;
   Push_Error, Pop_Error : exception;
end Stack_Object_Package;
```

- MONITORS ARE DIFFERENT:
  - NO USE FOR ENTRIES TO REPORT WHETHER OTHER ENTRIES' PRECONDITIONS HOLD.
    - ANOTHER TASK CAN INVALIDATE THE PRECONDITION BETWEEN THE TIME IT IS TESTED AND THE TIME THE OPERATION IS INVOKED.
    - EXAMPLE: TASK 1 CONFIRMS STACK IS NONEMPTY, TASK 2 POPS THE LAST ITEM IN THE STACK, TASK 1 TRIES TO POP THE STACK IT JUST EXAMINED.
  - RAISING AN EXCEPTION WOULD BE INAPPROPRIATE SINCE OTHER TASKS CAN ROUTINELY AND RANDOMLY INVALIDATE PRECONDITIONS.
  - SOLUTION: USE GUARDS SO THAT AN ENTRY CALL IS NOT ACCEPTED WHEN ITS PRECONDITION DOES NOT HOLD.
  - AN ENTRY CALL ISSUED WHEN ITS PRECONDITION IS FALSE WAITS UNTIL OPERATIONS BY OTHER TASKS MAKE THE PRECONDITION TRUE.

INSTRUCTOR NOTES

THIS IS THE MONITOR EQUIVALENT OF THE PACKAGE ON THE PREVIOUS SLIDE. POINT OUT THE LACK OF EXCEPTIONS, THE LACK OF OPERATIONS TO DETERMINE WHETHER A STACK IS EMPTY OR FULL, AND THE GUARDS.

USING GUARDS TO ENFORCE PRECONDITIONS - EXAMPLE

```
task Shared_Stack is
   entry Push (Item : in Integer);
   entry Pop  (Item : out Integer);
end Shared_Stack;

task body Shared_Stack is
   Stack_Size : constant := 100;
   Elements   : array (1 .. Stack_Size) of Integer;
   Top        : Integer range 0 .. Stack_Size := 0;
begin
   loop
      select
         when Top < Stack_Size =>
            accept Push (Item : in Integer) do
               Top := Top + 1;
               Elements (Top) := Item;
            end Push;
      or
         when Top > 0 =>
            accept Pop (Item : out Integer) do
               Item := Elements (Top);
            end Pop;
            Top := Top - 1;
      or
         terminate;
      end select;
   end loop;
end Shared_Stack;
```

INSTRUCTOR NOTES

DISCUSS THE SOLUTION FOR PART 1 BEFORE GOING ON TO PART 2.

IN PART 2, TELL STUDENTS TO PLACE COMPLICATED PARTS OF THE COMPUTATION IN SUBPROGRAMS AND WRITE BODY STUBS FOR THOSE SUBPROGRAMS (E.G. THE HANDLING OF A CALL ON Find_And_Reserve_Request -- SEE SOLUTION). SUBUNITS SHOULD BE WRITTEN ONLY IF THERE IS TIME. THE IDEA IS TO EMPHASIZE MONITOR-WRITING SKILLS, NOT TABLE-SEARCHING SKILLS.

A SUGGESTED ALLOCATION OF TIME IS 20 MINUTES FOR SOLUTION AND DISCUSSION OF PART 1, 55 MINUTES FOR SOLUTION AND DISCUSSION OF PART 2, BUT USE YOUR OWN DISCRETION.

EXERCISE 12.1

A 36-FLOOR BUILDING IS SERVED BY 5 ELEVATORS CONTROLLED BY A COMPUTER. EACH FLOOR HAS AT MOST ONE "UP" CALL BUTTON AND ONE "DOWN" CALL BUTTON. A PICKUP REQUEST MAY BE SERVICED BY ANY OF THE ELEVATORS.

THE CONTROL PROGRAM MAINTAINS A TABLE OF PENDING PICKUP REQUESTS, USED AS FOLLOWS:

- FOR EACH FLOOR, A REQUEST FOR SERVICE IN A PARTICULAR DIRECTION IS EITHER ABSENT, PENDING, OR BEING SERVICED.
- A TASK MONITORING CALL BUTTONS ATTEMPTS TO ENTER REQUESTS IN THE TABLE. IF THE CORRESPONDING TABLE ENTRY WAS MARKED ABSENT, IT IS CHANGED TO PENDING. OTHERWISE THE ATTEMPT TO ENTER A REQUEST HAS NO EFFECT.
- A TASK CONTROLLING AN ELEVATOR SEARCHES FOR A PENDING REQUEST, SPECIFYING AN "UP" OR "DOWN" REQUEST AND THE FIRST AND LAST FLOOR TO BE SCANNED. A SCAN IN EITHER DIRECTION MAY BE SPECIFIED (E.G. FROM 3 UPWARD TO 7 OR FROM 7 DOWNWARD TO 3): THE FIRST PENDING REQUEST ENCOUNTERED IS USED. THE SEARCH YIELDS A BOOLEAN VALUE INDICATING WHETHER A PENDING REQUEST WAS FOUND AND A FLOOR NUMBER INDICATING WHERE THE REQUEST CAME FROM (SET TO 1 IF NO REQUEST WAS FOUND).
- A TASK CONTROLLING AN ELEVATOR ALWAYS HANDLES ANY PENDING REQUEST IT FINDS. IT MARKS THE REQUEST AS BEING SERVICED SO THAT NO OTHER ELEVATOR TRIES TO HANDLE IT.
- ONCE THE ELEVATOR HAS ARRIVED AT THE REQUESTING FLOOR AND IS ABOUT TO LEAVE IT, ITS CONTROLLING TASK MARKS THE REQUEST AS ABSENT, SO THAT NEW REQUESTS FOR THAT FLOOR AND DIRECTION CAN BE ENTERED.

THE FOLLOWING PACKAGE HIDES THE TABLE AND PROVIDES AN INTERFACE TO IT:

```
package Request_Table_Package is
   type Floor_Type is range 1 .. 36;
   type Direction_Type is (Up, Down);
   -- [Declarations of operations on the table]
end Request_Table_Package;
```

THIS EXERCISE HAS TWO PARTS:

1. COMPLETE THE PACKAGE SPECIFICATION.
2. WRITE THE PACKAGE BODY.

INSTRUCTOR NOTES

ALLOW 45 MINUTES FOR THIS SECTION.

Section 13

MESSAGE BUFFERS

INSTRUCTOR NOTES

BULLET 3 GIVES THE MAIN MESSAGE OF SECTION 13.

RENDEZVOUS AS BUILDING BLOCKS

- RENDEZVOUS ARE THE BASIC MECHANISM BY WHICH TWO Ada TASKS COMMUNICATE.
  - ONE TASK CALLS ANOTHER TASK'S ENTRY
  - THE OTHER TASK ACCEPTS A CALL ON THAT ENTRY
- SOMETIMES THIS MECHANISM IS TOO INFLEXIBLE.
  - A TASK SENDING INFORMATION MUST WAIT IF A TASK RECEIVING INFORMATION IS NOT READY FOR A RENDEZVOUS.
- RENDEZVOUS CAN BE USED AS BUILDING BLOCKS TO BUILD MORE POWERFUL COMMUNICATIONS MECHANISMS.
  - SPECIAL NEW TASKS ARE INTRODUCED, AND TWO TASKS COMMUNICATE WITH EACH OTHER INDIRECTLY BY RENDEZVOUSING WITH THESE SPECIAL TASKS.
- MESSAGE BUFFERS ARE SUCH A MECHANISM.
  - THEY ALLOW INFORMATION TO BE SENT BEFORE THE RECIPIENT IS READY TO RECEIVE IT.
  - THE SENDING TASK NEED NOT WAIT FOR THE INFORMATION TO BE RECEIVED.

INSTRUCTOR NOTES

THIS IS A SPECIFIC INSTANCE OF A GENERAL PROBLEM:

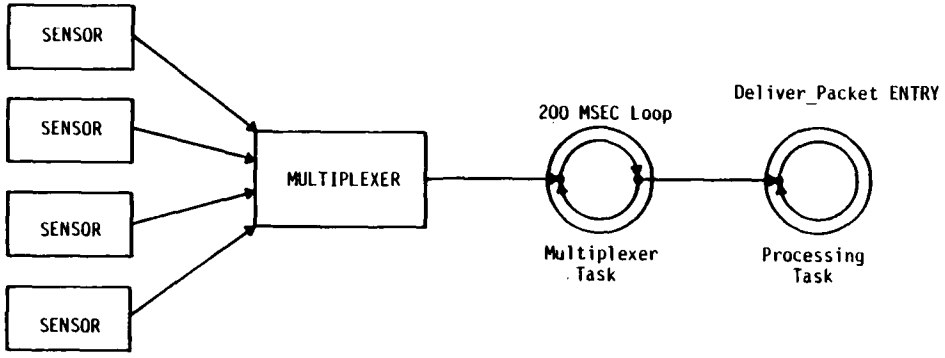
WHEN ONE TASK PRODUCES DATA THAT ANOTHER TASK CONSUMES, THE CONSUMING TASK MAY FALL BEHIND THE PRODUCING TASK, SLOWING IT DOWN.

- BULLET 2: THE PACKET CONSISTS OF ONE READING FROM EACH SENSOR.

- BULLET 3: THE MULTIPLEXER TASK IS RESPONSIBLE FOR READING FROM THE MULTIPLEXER AT NEARLY EXACT 200 MILLISECOND INTERVALS, SO EACH PACKET ASSEMBLED IS READ ONCE AND ONLY ONCE.

- BULLET 5: THE PROCESSING TASK FALLS BEHIND ONLY OCCASIONALLY, AND ONLY BY A LITTLE BIT, BUT THIS IS ENOUGH TO THROW OFF THE CRITICAL TIMING OF THE MULTIPLEXER TASK.

A COMMON PROBLEM



- SEVERAL SENSORS CONNECTED TO A MULTIPLEXER.
- EVERY 200 MILLISECONDS, THE MULTIPLEXER ASSEMBLES A PACKET OF SENSOR READINGS.
- A MULTIPLEXER TASK READS FROM THE MULTIPLEXER ONCE EVERY 200 MILLISECONDS AND CALLS THE Deliver_Packet ENTRY OF A PROCESSING TASK.
  - CONTENTS OF PACKET PASSED AS A PARAMETER
- THE PROCESSING TASK HAS A LOOP THAT REPEATEDLY ACCEPTS A CALL ON THE Deliver_Packet ENTRY AND PROCESSES THE PACKET DELIVERED.

- THE PROBLEM:
  - THE PROCESSING TASK OCCASIONALLY TAKES A LITTLE MORE THAN 200 MILLISECONDS TO PROCESS A PACKET.
  - MORE THAN 200 MILLISECONDS GO BY WITHOUT A CALL ON Deliver_Packet BEING ACCEPTED, SO THE MULTIPLEXER TASK IS FORCED TO WAIT AT AN ENTRY CALL.
  - SINCE THE MULTIPLEXER TASK GOES MORE THAN 200 MILLISECONDS WITHOUT READING, ONE OF THE PACKETS ASSEMBLED BY THE MULTIPLEXER IS NEVER SEEN.

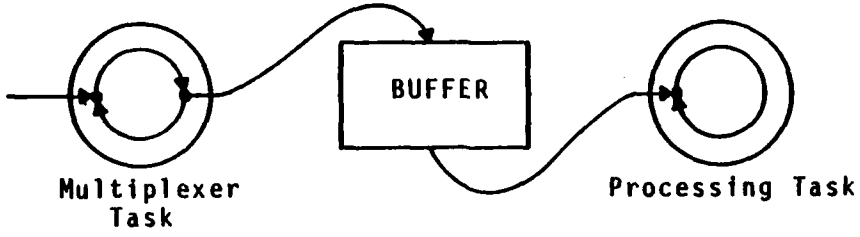
INSTRUCTOR NOTES

- BULLET 3: IF THE MULTIPLEXER TASK PRODUCES THREE PACKETS BEFORE THE PROCESSING TASK COMPLETES PROCESSING THE FIRST, THE EFFECT WILL BE LIKE THAT DESCRIBED ON THE PREVIOUS SLIDE. THE MULTIPLEXER TASK WILL HAVE TO WAIT UNTIL THE PROCESSING TASK TAKES THE SECOND PACKET FROM THE BUFFER TO MAKE ROOM FOR THE THIRD PACKET. HOWEVER, IF THE PROCESSING TASK TAKES MORE THAN 200 MILLISECONDS ONLY RARELY, AND NEVER MISSES BY MUCH, IT IS NOT LIKELY TO FALL THIS FAR BEHIND.

  LATER, WE SHALL PRESENT A SOLUTION (n-ELEMENT MESSAGE BUFFERS) THAT IS MORE APPROPRIATE WHEN GREAT VARIATION IN PROCESSING TIMES IS POSSIBLE.

- BULLET 4: THE TERM MESSAGE MEANS WHATEVER KIND OF DATA IS BEING PRODUCED BY ONE TASK, SAVED IN THE BUFFER, AND CONSUMED BY ANOTHER TASK. IN THIS EXAMPLE, A PACKET IS A MESSAGE.

SOLUTION: A BUFFER



- THE MULTIPLEXER TASK POLLS AT 200 MILLISECOND INTERVALS AND DROPS EACH PACKET INTO A BUFFER. THE BUFFER CAN HOLD ONE PACKET.

- THE PROCESSING TASK FINDS ITS DATA IN THE BUFFER.

  - IF DATA IS ALREADY THERE, THE PROCESSING TASK READS THE PACKET FROM THE BUFFER AND BEGINS PROCESSING.
  - IF THE BUFFER IS EMPTY, THE PROCESSING TASK WAITS UNTIL DATA ARRIVES, THEN TAKES IT OUT IMMEDIATELY AND BEGINS PROCESSING IT.

- AS LONG AS THE AVERAGE PROCESSING TIME IS UNDER 200 MILLISECONDS AND THE PROCESSING TASK DOES NOT DEVIATE MUCH FROM THIS AVERAGE, THE PROCESSING TASK DOES NOT SLOW DOWN THE MULTIPLEXER TASK.

- THIS BUFFER IS CALLED A ONE-ELEMENT MESSAGE BUFFER.

INSTRUCTOR NOTES

THIS IS THE CLASSICAL IMPLEMENTATION OF A MONITOR, AS DESCRIBED IN SECTION 12. IN THIS PARTICULAR CASE, THE TASK BODY CAN BE MADE SIMPLER. THE NEXT SLIDE EXPLAINS HOW.

BECAUSE MESSAGE BUFFERS ARE A COMMON BUILDING BLOCK IN MANY CONCURRENT SYSTEM DESIGNS, WE ENCLOSE THE MONITOR IN A GENERIC PACKAGE THAT CAN BE INSTANTIATED WITH ANY TYPE OF MESSAGE.

IMPLEMENTATION OF A ONE-ELEMENT MESSAGE BUFFER

- SINCE THE BUFFER IS A DATA OBJECT SHARED BY THE SENDING TASK AND THE RECEIVING TASK, IT SHOULD BE WRITTEN AS A MONITOR.
- GENERIC ONE-ELEMENT MESSAGE BUFFER:

```
generic
   type Message_Type is private;
package One_Element_Message_Buffer_Template is
   task type Message_Buffer_Type is
       entry Send (Message : in Message_Type);
       entry Receive (Message : out Message_Type);
   end Message_Buffer_Type;
end One_Element_Message_Buffer_Template;

package body One_Element_Message_Buffer_Template is
   task body Message_Buffer_Type is
       Buffer      : Message_Type;
       Buffer_Full : Boolean := False;
   begin
       loop
          select
              when not Buffer_Full =>
                 accept Send (Message : in Message_Type) do
                     Buffer := Message;
                 end Send;
                 Buffer_Full := True;
          or
              when Buffer_Full =>
                 accept Receive (Message : out Message_Type) do
                     Message := Buffer;
                 end Receive;
                 Buffer_Full := False;
          or
                 terminate;
          end select;
       end loop;
   end Message_Buffer_Type;
end One_Element_Message_Buffer_Template;
```

INSTRUCTOR NOTES

- BULLET 1: THIS LOOP IS LIFTED DIRECTLY FROM THE TASK BODY ON THE PREVIOUS SLIDE.

- BULLET 2: THE GUARDS ALLOW ONLY ONE OF THE TWO ACCEPT ALTERNATIVES TO BE CHOSEN, DEPENDING ON THE VALUE OF Buffer_Full. EACH ACCEPT ALTERNATIVE ENDS BY FLIPPING THE VALUE OF Buffer_Full FOR THE NEXT ITERATION.

- BULLET 3: THE LOOP IS WAITING AT THE SELECT STATEMENT WHEN THE BUFFER IS EMPTY AND AT THE ACCEPT STATEMENT FOR Receive WHEN THE BUFFER IS FULL. WE HAVE NOT ALLOWED FOR THE POSSIBILITY OF ACCEPTING A TERMINATE ALTERNATIVE WHEN THE BUFFER IS FULL.

IN A CORRECT SYSTEM DESIGN, A RECEIVING TASK SHOULD NOT BE ABLE TO TERMINATE WHILE MESSAGES SENT TO IT HAVE NOT YET BEEN RECEIVED.

AS LONG AS THE RECEIVING TASK IS STILL EXECUTING, ANY TASK IT IS ABLE TO CALL - INCLUDING A MESSAGE BUFFER TASK - WILL BE UNABLE TO ACCEPT A TERMINATE ALTERNATIVE. THEREFORE, WE MAY ASSUME THAT A TERMINATE ALTERNATIVE IS NEVER ACCEPTED WHEN THE BUFFER IS FULL.

A SIMPLIFICATION

- LOOP FROM Message_Buffer_Type TASK BODY:

```
loop
   select
       when not Buffer_Full =>
          accept Send (Message : in Message_Type) do
              Buffer := Message;
          end Send;
          Buffer_Full := True;
   or
       when Buffer_Full =>
          accept Receive (Message : out Message_Type) do
              Message := Buffer;
          end Receive;
          Buffer_Full := False;
   or
          terminate;
   end select;
end loop;
```

- OBSERVATION: Send CAN ONLY BE ACCEPTED ON ODD-NUMBERED ITERATIONS OF THE LOOP AND Receive CAN ONLY BE ACCEPTED ON EVEN-NUMBERED ITERATIONS.

- THE TASK BODY CAN BE REWRITTEN SO THAT THE INFORMATION IN Buffer_Full IS REFLECTED INSTEAD BY THE CURRENT POSITION IN THE LOOP:

```
loop
   select
       accept Send (Message : in Message_Type) do
          Buffer := Message;
       end Send;
   or
       terminate;
   end select;
   accept Receive (Message : out Message_Type) do
       Message := Buffer;
   end Receive;
end loop;
```

INSTRUCTOR NOTES

NOW A RECEIVING TASK MUST FALL n MESSAGES BEHIND A SENDING TASK TO FORCE IT TO WAIT.

IMPLEMENTATIONS OF QUEUES WERE DESCRIBED IN MODULE L305.

n-ELEMENT MESSAGE BUFFERS

- IF THERE CAN BE GREAT VARIATION IN THE AMOUNT OF TIME BETWEEN TWO Send OPERATIONS OR BETWEEN TWO Receive OPERATIONS, THE BUFFER MAY STILL BE FULL WHEN IT IS TIME TO SEND ANOTHER MESSAGE.
  - SENDING TASK MUST WAIT.
  - SAME PROBLEMS ARISE AS WITH NO BUFFER, ONLY LESS FREQUENTLY.
- MESSAGE BUFFER CAN BE ENHANCED SO THAT THE MONITOR PROTECTS A QUEUE OF MESSAGES INSTEAD OF A SINGLE MESSAGE.
  - A Send OPERATION PUTS A MESSAGE AT THE BACK OF THE QUEUE.
  - A Receive OPERATION REMOVES A MESSAGE FROM THE FRONT OF THE QUEUE.
- THE RECEIVING TASK MAY MOMENTARILY FALL FURTHER BEHIND THE SENDING TASK, WITHOUT FORCING THE SENDING TASK TO WAIT.
  - ALL UNRECEIVED MESSAGES ARE QUEUED.
  - MESSAGES ARE RECEIVED IN THE ORDER THEY ARE SENT.
- n-ELEMENT MESSAGE BUFFERS ARE SOMETIMES CALLED MESSAGE QUEUES.

INSTRUCTOR NOTES

AS BEFORE, WE WILL PRESENT A GENERIC IMPLEMENTATION OF MESSAGE BUFFERS.

THIS TIME, THERE IS A SECOND GENERIC PARAMETER - THE CAPACITY OF THE QUEUE.

THE PACKAGE BODY IS GIVEN ON THE NEXT SLIDE.

GENERIC DECLARATION FOR n-ELEMENT MESSAGE BUFFER

```
generic

   type Message_Type is private;
   Buffer_Capacity:  in Positive;

package Message_Buffer_Template is

   task type Message_Buffer_Type is
       entry Send (Message : in Message_Type);
       entry Receive (Message : out Message_Type);
   end Message_Buffer_Type;

end Message_Buffer_Template;
```

INSTRUCTOR NOTES

THE QUEUE IS IMPLEMENTED AS A CIRCULAR LIST.

MESSAGES ARE ADDED TO THE END OF BUFFER AND REMOVED FROM THE BEGINNING. THE LAST COMPONENT OF BUFFER IS TREATED AS THOUGH IT IS FOLLOWED IMMEDIATELY BY THE FIRST.

CIRCULAR LISTS WERE DESCRIBED IN MODULE L305.

Next_Insertion INDEXES THE EMPTY COMPONENT OF BUFFER IN WHICH THE NEXT MESSAGE WILL BE PLACED. Next_Removal INDEXES THE FULL COMPONENT OF BUFFER FROM WHICH THE NEXT MESSAGE WILL BE REMOVED (OR EQUALS Next_Insertion WHEN THE BUFFER IS EMPTY). THIS CONVENTION ALLOWS EACH ACCEPT STATEMENT TO CONTAIN A SINGLE ASSIGNMENT STATEMENT. Current_Size GIVES THE NUMBER OF FULL COMPONENTS.

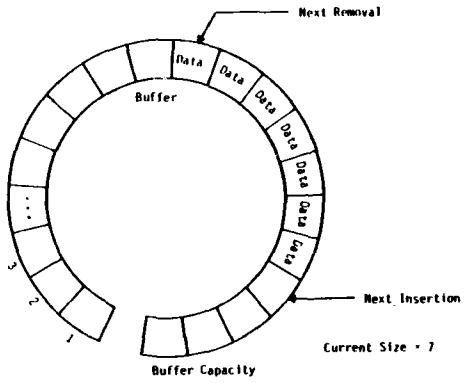
THE GUARDS PREVENT INSERTION INTO A FULL BUFFER OR REMOVAL FROM AN EMPTY BUFFER.

GENERIC BODY FOR n-ELEMENT MESSAGE BUFFERS

```
package body Message_Buffer_Template is

   task body Message_Buffer_Type is
      subtype Buffer_Index_Subtype is Integer range 1 .. Buffer_Capacity;
      Buffer       : array (Buffer_Index_Subtype) of Message_Type;
      Current_Size : Integer range 0 .. Buffer_Capacity := 0;
      Next_Insertion, Next_Removal: Buffer_Index_Subtype := 1;
   begin
      loop
         select
            when Current_Size < Buffer_Capacity =>
               accept Send (Message : In Message_Type) do
                  Buffer (Next_Insertion) := Message;
               end Send;
               if Next_Insertion = Buffer'Last then
                  Next_Insertion := 1;
               else
                  Next_Insertion := Next_Insertion + 1;
               end if;
               Current_Size := Current_Size + 1;
         or
            when Current_Size> 0 =>
               accept Receive (Message : out Message_Type) do
                  Message := Buffer (Next_Removal);
               end Receive;
               if Next_Removal = Buffer'Last then
                  Next_Removal := 1;
               else
                  Next_Removal := Next_Removal + 1;
               end if;
               Current_Size := Current_Size - 1;
         or
               terminate;
         end select;
      end loop;
   end Message_Buffer_Type;

end Message_Buffer_Template;
```



INSTRUCTOR NOTES

- BULLET 1: LINKED LISTS WERE DESCRIBED IN MODULE L305.

- BULLET 3: LARGER BUFFERS CANNOT MAKE UP FOR DIFFERENCES IN AVERAGE SPEED, BUT ONLY FOR GREATER MOMENTARY DEVIATIONS FROM THE LONG-TERM AVERAGE. IN THE LONG RUN, THE CONSUMING TASK MUST BE ABLE TO CONSUME MESSAGES FASTER THAN THE PRODUCING TASK CAN PRODUCE THEM.

UNBOUNDED MESSAGE BUFFERS

- THE QUEUE INSIDE THE Message_Buffer_Type TASK BODY CAN BE IMPLEMENTED WITH A LINKED LIST RATHER THAN AN ARRAY.

- THIS PROVIDES MESSAGE BUFFERS WITH VIRTUALLY UNBOUNDED CAPACITY.

  - ONLY LIMITATION IS AMOUNT OF STORAGE AVAILABLE FOR ALLOCATION OF LIST CELLS.

- THIS ISN'T AS USEFUL AS IT SOUNDS.

  - LET s BE THE AVERAGE AMOUNT OF TIME IT TAKES THE SENDING TASK TO PRODUCE A MESSAGE.

    Let r BE THE AVERAGE AMOUNT OF TIME IT TAKES THE RECEIVING TASK TO PROCESS A MESSAGE.

  - IF s > r, THE PROBABILITY OF A BOUNDED QUEUE BECOMING FULL DECREASES EXPONENTIALLY WITH QUEUE SIZE.

  - IF s < r, EVEN AN UNBOUNDED QUEUE IS USELESS.

    - MESSAGE ARE CONTINUALLY PRODUCED FASTER THAN THEY ARE PROCESSED.
    - THE RECEIVING TASK WILL NEVER CATCH UP.
    - SOME MESSAGES WILL NEVER BE READ.
    - THIS IS A SYMPTOM OF A SERIOUS DESIGN PROBLEM.

INSTRUCTOR NOTES

THIS SECTION BEGAN BY EXPLAINING THAT RENDEZVOUS CAN BE USED AS BUILDING BLOCKS TO CONSTRUCT OTHER COMMUNICATIONS MECHANISMS.

THE SECTION CONCLUDES BY POINTING OUT THAT RENDEZVOUS ARE NOT THE ONLY WAY TO DO THIS. IF NECESSARY, MESSAGE BUFFERS CAN BE IMPLEMENTED DIRECTLY IN HARDWARE (PERHAPS USING A TEST-AND-SET MACHINE INSTRUCTION TO ENSURE MUTUAL EXCLUSION).

EMPHASIZE THAT WE DO NOT MEAN TO SUGGEST THAT RENDEZVOUS ARE NECESSARILY INEFFICIENT.

VERY FAST MESSAGE BUFFERS

- SOME PROGRAM DESIGNS USE MESSAGE BUFFERS FOR ALL INTERTASK COMMUNICATION.
- EFFICIENT SERVICING OF Send AND Receive OPERATIONS MAY BE CRITICAL.
- HARDWARE MAY SUPPORT A MORE EFFICIENT IMPLEMENTATION THAN MONITORS.
- Send AND Receive OPERATIONS CAN BE IMPLEMENTED WITH CODE PROCEDURES.
  - Send AND Receive PROCEDURES DECLARED IN A PACKAGE.
  - CODE PROCEDURES HIDDEN IN PACKAGE BODY.
  - MUTUAL EXCLUSION PROVIDED BY HARDWARE OR RUNTIME SYSTEM.
- NOT RECOMMENDED AS THE USUAL PRACTICE, BUT THE OPTION IS AVAILABLE WHEN NEEDED.

INSTRUCTOR NOTES

ALLOW 60 MINUTES FOR THIS SECTION AND FOLLOW IT WITH A BREAK.

Section 14

CYCLIC PROCESSING

INSTRUCTOR NOTES

- BULLET 2:

  A DETAILED EXAMINATION OF THESE CONSIDERATIONS IS BEYOND THE SCOPE OF THIS COURSE. THE MAIN POINT IS THAT MINIMUM FREQUENCY REQUIREMENTS ARE EXTERNALLY IMPOSED, NOT SUBJECT TO THE DISCRETION OF THE SOFTWARE DESIGNER OR PROGRAMMER.

- BULLET 3:

  THE NEXT FIVE SLIDES DESCRIBE TRADITIONAL CYCLIC EXECUTIVES AND PROVIDE COMMON TERMINOLOGY AND A BASIS FOR COMPARISON.

  THE WORD "TASK" AS USED IN CYCLIC EXECUTIVES DOES NOT CORRESPOND TO THE MEANING OF THAT WORD IN Ada. BE SURE THAT STUDENTS FAMILIAR WITH CYCLIC EXECUTIVES UNDERSTAND THE DISTINCTION.

- CYCLIC EXECUTIVES ARE DISCUSSED IN GREATER DETAIL IN "DESIGNING REAL TIME SYSTEMS IN Ada", FINAL REPORT, 1986 CHAPTER 2.

THE NEED FOR CYCLIC PROCESSING

- IN TYPICAL REAL-TIME APPLICATIONS (e.g. AVIONICS AND GUIDANCE), CERTAIN ACTIONS MUST BE PERFORMED REPETITIVELY, AT SPECIFIED INTERVALS.
  - DATA SAMPLING
  - CONTROL (FEEDBACK) LOOPS
- INTERVALS ARE BASED ON THE NATURE OF THE APPLICATION AND SYSTEMS CONTROL THEORY.
  - IF DATA BEING SAMPLED FLUCTUATES AT SOME FREQUENCY, IT SHOULD BE SAMPLED AT LEAST TWICE THAT FREQUENCY TO AVOID ERRONEOUS EXTRAPOLATIONS (ALIASING).
  - "TRANSPORT LAG" BETWEEN FEEDBACK AND OUTPUT IN A CONTROL LOOP MUST BE KEPT SMALL, OR FEEDBACK WILL BE OUT OF PHASE. CONTROL LOOP CAN BECOME UNSTABLE.
- TYPICAL SOLUTION IS A CYCLIC EXECUTIVE.
  - PROCESSING "TASKS" ACTIVATED IN A FIXED ORDER AT A FIXED FREQUENCY.
  - THESE "TASKS" ARE SMALL NONCONCURRENT PROCESSING STEPS.
  - WE'LL CALL THESE "TASKS" ACTIVITIES TO AVOID CONFUSION.

INSTRUCTOR NOTES

THIS IS A DESCRIPTION OF HOW SIMPLE CYCLIC EXECUTIVES ARE ORGANIZED. AS SLIDE 14-6 WILL EXPLAIN, PURE CYCLIC EXECUTIVES ARE TOO SIMPLE MINDED FOR MOST REAL-TIME SYSTEMS.

THE NEXT SLIDE GIVES AN ILLUSTRATION OF MAJOR AND MINOR CYCLES.

- BULLET 1:

  - ITEM 2: FREQUENCIES IMPLEMENTED BY A CYCLIC EXECUTIVE ARE TYPICALLY POWERS OF TWO TIMES THE MINOR CYCLE FREQUENCY. (THIS MAKES IT EASY TO DETERMINE WHETHER A GIVEN ACTIVITY SHOULD BE PERFORMED DURING A GIVEN MINOR CYCLE BY EXAMINING PART OF THE BINARY REPRESENTATION OF THE CURRENT MINOR CYCLE NUMBER. AN ACTIVITY TO BE PERFORMED EVERY 2^n MINOR CYCLES COULD BE ACTIVATED ONLY FOR MINOR CYCLE NUMBERS IN WHICH THE n LOW-ORDER BITS HAVE A SPECIFIED VALUE. IN ADDITION, THE RESTRICTION TO POWERS OF TWO MAKES IT EASY TO SYNCHRONIZE ACTIVITIES AT DIFFERENT FREQUENCIES.) AN ACTIVITY IS PERFORMED AT THE LOWEST OF THE IMPLEMENTED FREQUENCIES THAT IS GREATER THAN OR EQUAL TO ITS REQUIRED FREQUENCY.

    THE MINOR CYCLE FREQUENCY IS BASED ON THE HIGHEST FREQUENCY REQUIREMENT AND THE OTHER IMPLEMENTED FREQUENCIES ARE BASED ON THE MINOR CYCLE FREQUENCY.

MAJOR AND MINOR CYCLES

- MINOR CYCLE IS A LOOP EXECUTED AT REGULAR INTERVALS.
  - INTERVAL IS SMALL ENOUGH TO ACCOMMODATE THE HIGHEST REQUIRED PROCESSING FREQUENCY.
  - DEPENDING ON MINIMUM FREQUENCY REQUIRED BY THE PROBLEM, CERTAIN ACTIVITIES ARE PERFORMED EVERY 1, 2, 4, 8, ... MINOR CYCLES.
  - DIFFERENT ACTIVITIES PERFORMED IN DIFFERENT MINOR CYCLES.
- MAJOR CYCLE CONSISTS OF SOME PREDETERMINED NUMBER OF MINOR CYCLES.
  - IF LOWEST-FREQUENCY ACTIVITY IS PERFORMED ONCE EVERY 64 MINOR CYCLES, A MAJOR CYCLE MIGHT CONSIST OF 64 MINOR CYCLES.
  - EACH MAJOR CYCLE CONSISTS OF THE SAME SEQUENCE OF ACTIONS.
  - PROGRAM EXECUTION CONSISTS OF REPEATED EXECUTION OF THE MAJOR CYCLE.

INSTRUCTOR NOTES

- BULLET 1: THESE ARE THE ASSUMED EXTERNAL FREQUENCY REQUIREMENTS.

- BULLET 2: THIS SHOWS THE ASSIGNMENT OF EACH ACTIVITY TO THE LOWEST IMPLEMENTED FREQUENCY GREATER THAN OR EQUAL TO THE REQUIRED FREQUENCY.

  THE MINOR CYCLE TIME OF 4 MILLISECONDS CORRESPONDS TO THE HIGHEST REQUIRED FREQUENCY - THAT OF Activity_A.

  SOMETIMES AN ACTIVITY IS TOO LONG TO FIT INTO ONE MINOR CYCLE AND MUST BE SPLIT INTO PIECES EXECUTED IN SUCCESSIVE MINOR CYCLES. THIS IS ILLUSTRATED LATER, BUT NOT ON THIS SLIDE.

  NOTE: AN ACTIVITY PERFORMED (FOR EXAMPLE) 62.5 TIMES A SECOND IS SOMETIMES DESCRIBED AS OPERATING AT 62.5 Hertz (62.5 Hz).

- BULLET 3: THE MAJOR CYCLE IS BASED ON THE IMPLEMENTED FREQUENCY USED BY THE LOWEST FREQUENCY ACTIVITY. Activity_E IS PERFORMED ONCE EVERY 8 MINOR CYCLES.

- BULLET 4: THE BOTTOM ROW SHOWS THE DIVISION OF THE MAJOR CYCLE INTO 8 MINOR CYCLES EACH LASTING 4 MILLISECONDS. DIFFERENT COMBINATIONS OF ACTIVITIES ARE SCHEDULED FOR EACH MINOR CYCLE. ALL PROCESSING IS SEQUENTIAL. (IN THE FIRST MINOR CYCLE, FOR EXAMPLE, Activity_A IS EXECUTED, THEN Activity_B, THEN Activity_E. THEN THE SYSTEM WAITS UNTIL IT IS TIME FOR THE NEXT MINOR CYCLE.)

  THE FOUR ROWS LABELED "REQUIRED PROCESSING" SHOW ACTIVITIES THAT MUST BE PERFORMED ONCE WITHIN SPECIFIED SLICES OF THE MAJOR CYCLE. ALLOCATION OF AN ACTIVITY TO A PARTICULAR MINOR CYCLE WITHIN A SLICE IS DISCUSSED ON SLIDE 14-5.

EXAMPLE OF MAJOR AND MINOR CYCLES

- ACTIVITIES AND MINIMUM ALLOWABLE FREQUENCIES:

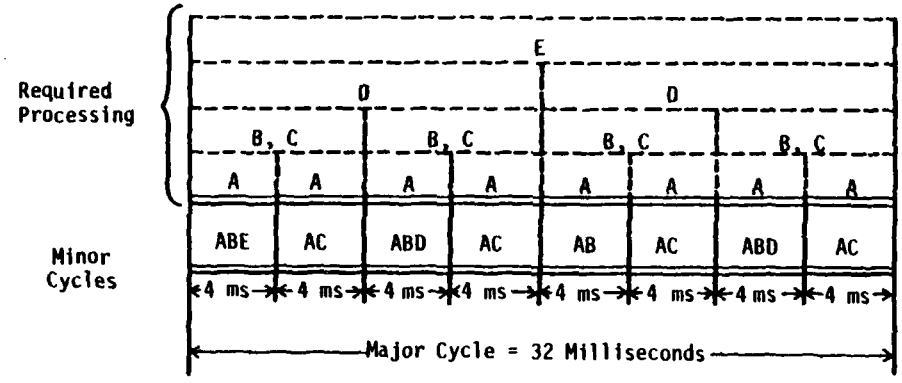
  ACTIVITY A: AT LEAST 250 TIMES A SECOND
  ACTIVITY B: AT LEAST 100 TIMES A SECOND
  ACTIVITY C: AT LEAST 75 TIMES A SECOND
  ACTIVITY D: AT LEAST 50 TIMES A SECOND
  ACTIVITY E: AT LEAST 30 TIMES A SECOND

- FREQUENCIES BASED ON 4 MILLISECOND MINOR CYCLE TIME:

  ACTIVITY A: ONCE EVERY MINOR CYCLE (250 TIMES A SECOND)
  ACTIVITY B: ONCE EVERY 2 MINOR CYCLES (125 TIMES A SECOND)
  ACTIVITY C: ONCE EVERY 2 MINOR CYCLES (125 TIMES A SECOND)
  ACTIVITY D: ONCE EVERY 4 MINOR CYCLES (62.5 TIMES A SECOND)
  ACTIVITY E: ONCE EVERY 8 MINOR CYCLES (31.25 TIMES A SECOND)

- MAJOR CYCLE CONSISTS OF 8 MINOR CYCLES.

- POSSIBLE SCHEDULING OF ACTIVITIES WITHIN A MAJOR CYCLE:



INSTRUCTOR NOTES

SIMULTANEOUS UPDATE IS NOT A PROBLEM BECAUSE THERE IS NO CONCURRENCY. ACCESS TO THE SHARED DATA OCCURS IN A PREDICTABLE ORDER AND CANNOT BE INTERRUPTED.

(REMEMBER, WE ARE DESCRIBING A PURE CYCLIC EXECUTIVE, WITH NO EVENT-DRIVEN PROCESSING.)

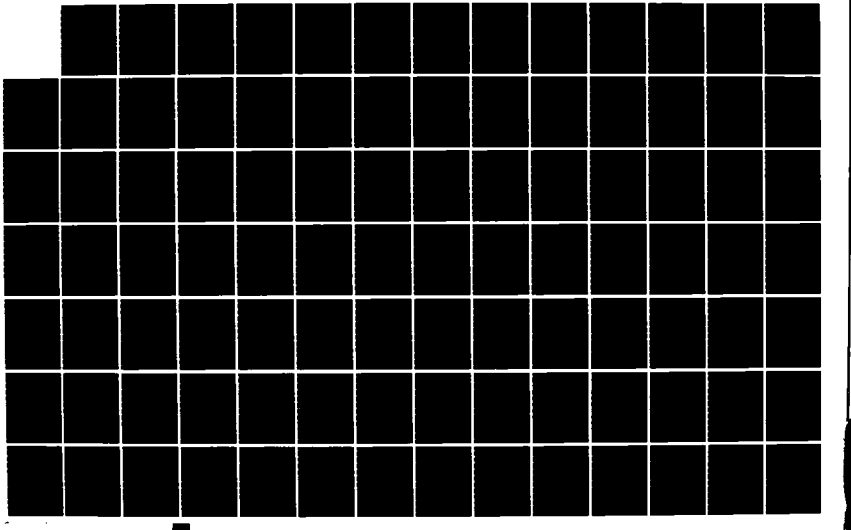
THE SCHEDULING OF ACTIVITIES MAY HAVE TO REFLECT DATA DEPENDENCIES. IF Activity_B SETS VARIABLES THAT Activity_C EXAMINES, Activity_B SHOULD BE SCHEDULED BEFORE Activity_C.

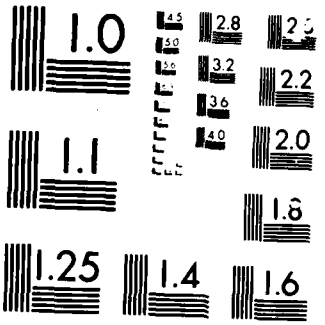
COMMUNICATION AMONG ACTIVITIES

- ACTIVITIES ARE RUN SEQUENTIALLY, IN AN ORDER MANDATED IN THE CYCLIC EXECUTIVE.
- SIMULTANEOUS UPDATE IS NOT A PROBLEM.
- ACTIVITIES COMMUNICATE BY SETTING AND EXAMINING SHARED (GLOBAL) VARIABLES.
- ORDER IN WHICH ACTIVITIES ARE SCHEDULED MAY BE CRUCIAL.

AD-A166 352 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS IN ADA L401 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 2/6

UNCLASSIFIED F/G 5/9 NL





MICROCOPY RESOLUTION TEST CHART

INSTRUCTOR NOTES

- BULLET 1:

  THESE ARE A FEW OF THE 32,768 WAYS IN WHICH ACTIVITIES SHOWN TWO SLIDES AGO COULD HAVE BEEN ALLOCATED TO MINOR CYCLES IN ACCORDANCE WITH THE PROCESSING REQUIREMENTS FOR VARIOUS SLICES OF THE MAJOR CYCLE. THE FIRST IS THE ALLOCATION THAT WAS DEPICTED EARLIER.

- BULLET 2:

  - ITEM 1: EACH OF THE POSSIBILITIES ILLUSTRATED EXECUTES Activity_A 8 TIMES, Activities_B and C 4 TIMES, Activity_D 2 TIMES, AND Activity_E 1 TIME PER MAJOR CYCLE.

  - ITEM 2: THE SECOND POSSIBILITY UNDER BULLET 1 SHOWS IRREGULARLY SCHEDULED INTERVALS. RATHER THAN BECOMING ACTIVE EVERY SECOND MINOR CYCLE, Activities_B and C TAKE TURNS BECOMING ACTIVE IN TWO CONSECUTIVE CYCLES THEN INACTIVE IN TWO CONSECUTIVE CYCLES. RATHER THAN BECOMING ACTIVE EVERY FOURTH MINOR CYCLE, Activity_D BECOMES ACTIVE IN TWO CONSECUTIVE CYCLES AND INACTIVE IN SIX CONSECUTIVE CYCLES. (CONSIDER TWO MAJOR CYCLES PLACED END-TO-END.)

  - ITEM 3: THE THIRD POSSIBILITY UNDER BULLET 1 SHOWS UNEVEN DISTRIBUTION OF PROCESSING LOAD. EVERY ACTIVITY PERFORMED IN MINOR CYCLE 1, EVERY ACTIVITY BUT E IN MINOR CYCLE 5, BUT ONLY Activity_A IN THE EVEN-NUMBERED CYCLES

  - ITEM 4: THE FOURTH POSSIBILITY UNDER BULLET 1 SHOWS HOW THE MAJOR CYCLE MIGHT HAVE BEEN DESIGNED IF Activity_E USED DATA PRODUCED BY ALL THE OTHER ACTIVITIES, ALL ACTIVITIES USED DATA PRODUCED BY Activity_D, AND Activity_B USED DATA PRODUCED BY Activity_C.

SCHEDULING OF ACTIVITIES WITHIN THE MAJOR CYCLE

- MANY POSSIBILITIES:

ABE	AC	ABD	AC	AB	AC	ABD	AC
CA	BA	BA	CDA	CDA	BAE	BA	CA
ABCDE	A	ABC	A	ABCD	A	ABC	A
DAC	AB	AC	AB	DAC	AB	AC	ABE

  - etc.

- CRITERIA:

  - REQUIRED FREQUENCY
  - REGULARITY OF INTERVALS
  - DISTRIBUTION OF PROCESSING LOAD
  - DATA DEPENDENCY

- DESIGNING THE MAJOR CYCLE IS A DIFFICULT MANUAL ACTIVITY.

INSTRUCTOR NOTES

MACLAREN'S ARTICLE (SEE BIBLIOGRAPHY) IDENTIFIES THREE LEVELS OF SCHEDULING COMPLEXITY:

> "LEVEL 1 - PURELY CYCLIC, NO ASYNCHRONOUS EVENTS OR VARIATIONS IN COMPUTING REQUIREMENTS. VERY SIMPLE CONTROL SYSTEMS, SUCH AS THOSE FOR SMALL MISSILES AND TARGET DRONES, OFTEN SATISFY THESE CONDITIONS.
>
> "LEVEL 2 - MOST CYCLIC, WITH SOME ASYNCHRONOUS EVENTS AND 'BURST' COMPUTING LOADS, SUCH AS FAULT RECOVERY, EXTERNAL COMMANDS, OR OPERATOR INTERFACES. 'NATURAL' PROCESSING FREQUENCIES FOR THE DIFFERENT FUNCTIONS MAY NOT BE CONVENIENT MULTIPLES OF EACH OTHER. THIS LEVEL IS TYPICAL OF MODERN AVIONICS AND SPACE SYSTEMS.
>
> "LEVEL 3 - ASYNCHRONOUS, OR EVENT-DRIVEN. CYCLIC PROCESSING DOES NOT DOMINATE. EXAMPLES OF SUCH APPLICATIONS WOULD INCLUDE COMMUNICATIONS, RADAR TRACKING, AND MOST GENERAL PURPOSE OPERATING SYSTEMS."

THUS TYPICAL AVIONICS, RADAR TRACKING, AND COMMUNICATIONS SYSTEMS DO NOT HAVE PURE CYCLIC EXECUTIVES OF THE KIND WE HAVE DESCRIBED.

AT HIGHER LEVELS, THE ASSUMPTIONS THAT MAKE THE CYCLIC EXECUTIVE A SIMPLE AND APPEALING MODEL BEGIN TO FALL APART.

COMPLICATIONS

- EXCEPT ON THE SIMPLEST SYSTEMS, THERE ARE ACTIVITIES THAT ARE PERFORMED ON A NONPERIODIC BASIS.
  - EVENT-DRIVEN ACTIVITIES LIKE INTERRUPT HANDLERS
  - "BURST" LOADS (TEMPORARY OVERLOAD)
  - BACKGROUND PROCESSING PERFORMED WHENEVER TIME IS AVAILABLE (e.g. SELF-TEST)
- ASYNCHRONISM COMPLICATES THE CONTROL STRUCTURE.
  - PROCESSING MAY OVERFLOW ITS ALLOTTED MINOR CYCLE.
  - CYCLES MAY HAVE TO EXPAND OR SHRINK.
  - DATA MAY HAVE TO BE BUFFERED.
- DEPARTURE FROM PREDETERMINED ORDER COMPLICATES COMMUNICATION.
  - INTERLEAVED ACCESS TO SHARED DATA BECOMES A PROBLEM.

INSTRUCTOR NOTES

- BULLET 1: THE DIRECT IMPLEMENTATION INVOLVES A SINGLE TASK EXECUTING A LOOP OF THE FOLLOWING FORM:

```
loop
   accept Timer_Interrupt;
   Minor_Cycle_Number := (Minor_Cycle_Number mod Number_Of_Minor_Cycles) + 1;
   case Minor_Cycle_Number is
       ...
   end case;
end loop;
```

  MACLAREN FEELS THAT THIS APPROACH IS APPROPRIATE, BECAUSE OF ITS FAMILIARITY, EFFICIENCY, AND SIMPLICITY, FOR SYSTEMS SIMPLE ENOUGH TO USE A PURE CYCLIC EXECUTIVE.

- BULLET 2: MACLAREN FEELS THAT THE MODEL DESCRIBED HERE BECOMES MORE APPROPRIATE AS SYSTEMS BECAME MORE ASYNCHRONOUS AND SCHEDULING BECOMES MORE COMPLEX. HIS ARTICLE DESCRIBES A SPECTRUM OF TASKING MODELS RANGING FROM THE PURE CYCLIC EXECUTIVE SKETCHED ABOVE TO A PURELY ASYNCHRONOUS SINGLE-THREAD APPROACH.

- BULLET 3: IN THE ABSENCE OF TIMING CONSTRAINTS, ANY RUNTIME SYSTEM WOULD EVENTUALLY PRODUCE CORRECT OUTPUTS WITH THE APPROACH WE DESCRIBE. HOWEVER, A RUNTIME SYSTEM OF THE KIND DESCRIBED IS NEEDED TO ENSURE TIMELY EXECUTION OF EACH ACTIVITY.

- A MORE DETAILED EXAMINATION OF CYCLIC EXECUTIVE IMPLEMENTATION IN Ada IS GIVEN IN "DESIGNING REAL TIME SYSTEMS IN Ada", FINAL REPORT, 1986 CHAPTERS 2, 5 AND 6.

CYCLIC PROCESSING IN Ada

- Ada DOESN'T PRECLUDE THE TRADITIONAL APPROACH.
  - CYCLIC SCHEDULING, DRIVEN BY TIMER INTERRUPTS, CAN BE IMPLEMENTED EASILY IN Ada.
  - THIS DOES NOT TAKE ADVANTAGE OF Ada's STRENGTHS
- THE PREFERRED Ada APPROACH INVOLVES ONE TASK FOR EACH INDEPENDENT ACTIVITY TO BE PERFORMED REPETITIVELY.
  - EACH TASK EXECUTES A LOOP AT THE REQUIRED FREQUENCY.
  - ONE ITERATION EXECUTES A DELAY STATEMENT THAT EXPIRES IN TIME FOR THE NEXT ITERATION.
  - RENDEZVOUS ARE USED FOR COMMUNICATION AND SYNCHRONIZATION AMONG TASKS.
  - EACH TASK CORRESPONDS TO A SINGLE CONCEPTUAL THREAD.
- SUCCESS OF THIS APPROACH DEPENDS ON CERTAIN ASSUMPTIONS ABOUT THE RUNTIME SYSTEM:
  - PREEMPTIVE SCHEDULING.
  - PREEMPTION MAY OCCUR WHEN A DELAY EXPIRES OR A RENDEZVOUS COMPLETES. (THE TASK THAT BECOMES UNBLOCKED MAY PREEMPT THE TASK THAT WAS EXECUTING.)

INSTRUCTOR NOTES

- BULLET 1:

  THIS IS THE SAME PROBLEM SHOWN EARLIER, BUT WITH NEW INFORMATION ADDED ABOUT THE AMOUNT OF TIME EACH ACTIVITY REQUIRES.

  EACH 4 ms MINOR CYCLE MUST INCLUDE EXECUTION OF Activity A (1.0 ms) AND - IF THE PROCESSING LOAD IS TO BE EVENLY DISTRIBUTED - EITHER B (1.1 ms) OR C (0.6 ms). THUS, IN ORDER TO FIT D (3.0 ms) INTO THE CYCLE, WE SPLIT IT INTO TWO PIECES, D_1 (1.3 ms) AND D_2 (1.7 ms) AND ALLOCATE EACH PIECE TO A DIFFERENT MINOR CYCLE. (SEE BULLET 2 ON SLIDE.) THIS KIND OF SPLITTING IS PERVASIVE IN REAL SYSTEMS.

- BULLET 2:

  THE SUMS OF THE ACTIVITY EXECUTION TIMES FOR THE EIGHT MINOR CYCLES ARE 3.6 ms, 1.6 ms, 3.4 ms, 3.3 ms, 2.1 ms, 1.6 ms, 3.4 ms, 3.3 ms.

- BULLET 3:

  EACH TASK EXECUTES A LOOP THAT REPEATEDLY PERFORMS AN ACTIVITY AND THEN DELAYS FOR THE AMOUNT OF TIME NEEDED TO ACHIEVE THE REQUIRED FREQUENCY. SLIDES 14-8 THROUGH 14-13 DESCRIBE HOW WE DETERMINE THE AMOUNT OF THE DELAY. NOTICE THAT WE AIM EXACTLY FOR THE REQUIRED FREQUENCY, NOT THE MINOR CYCLE FREQUENCY DIVIDED BY SOME POWER OF TWO. THESE LOOPS ENSURE THAT ACTIVITIES ARE PERFORMED AT THE CORRECT FREQUENCY, BUT DO NOT CONSTRAIN THE ORDER IN WHICH ACTIVITIES ARE PERFORMED. THAT PROBLEM IS ADDRESSED BY SLIDES 14-14 AND 14-15.

  AS THE LOOP BODY FOR D Task ILLUSTRATES, ACTIVITIES THAT ARE PART OF THE SAME CONCEPTUAL THREAD OCCUR IN SEQUENCE AS PART OF THE SAME LOOP. THIS MAKES IT MUCH EASIER TO UNDERSTAND THE INTERACTION OF PROGRAM COMPONENTS AND THE CORRESPONDENCE BETWEEN THE PROGRAM AND WHAT THE PROGRAM IS TRYING TO MODEL.

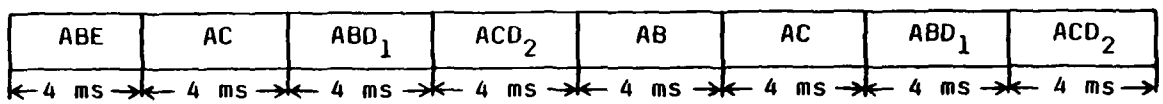
CYCLIC EXECUTIVE VERSUS SINGLE-THREAD APPROACH

- SAMPLE PROBLEM:

Activity	Minimum Frequency per second	Maximum Cycle Time (1 / Frequency)	Maximum Activity Execution Time
A	250 cycles	0.0040 sec	1.0 ms
B	100 cycles	0.0100 sec	1.1 ms
C	75 cycles	0.0133 sec	0.6 ms
D	50 cycles	0.0200 sec	3.0 ms
E	30 cycles	0.0333 sec	1.5 ms

Split into
D_1 (1.3 ms) and
D_2 (1.7 ms)

- CYCLIC EXECUTIVE MODEL:



- SINGLE-THREAD MODEL:

```
task body A_Task is
   ...
begin
   loop -- 250 times a second
     Activity_A;
     delay ... ;
   end loop;
end A_Task;
```

```
task body B_Task is
   ...
begin
   loop -- 100 times a second
     Activity_B;
     delay ... ;
   end loop;
end B_Task;
```

```
task body C_Task is
   ...
begin
   loop -- 75 times a second
     Activity_C;
     delay ... ;
   end loop;
end C_Task;
```

```
task body D_Task is
   ...
begin
   loop -- 50 times a second
     Activity_D1;
     Activity_D2;
     delay ... ;
   end loop;
end D_Task;
```

```
task body E_Task is
   ...
begin
   loop -- 30 times a second
     Activity_E;
     delay ... ;
   end loop;
end E_Task;
```

INSTRUCTOR NOTES

- BULLET 1:

  IF WE KNEW THAT Activity_A ALWAYS TOOK EXACTLY 1 ms OF CPU TIME AND THAT IT WAS TO BE REPEATED EVERY 4 ms, WE WOULD ALWAYS WANT TO DELAY FOR EXACTLY 3 ms.

- BULLET 2:

  HERE WE ATTEMPT TO MEASURE THE EXACT AMOUNT OF TIME IT TAKES TO PERFORM Activity_A EACH TIME AND SUBTRACT THAT FROM THE DESIRED CYCLE TIME TO OBTAIN THE LENGTH OF THE CORRESPONDING DELAY. ((CLOCK - Start_Time) SHOULD BE THE AMOUNT OF TIME THAT HAS PASSED SINCE THE ASSIGNMENT TO Start_Time.)

- BULLET 3:

  THE NEXT SLIDE DEPICTS THE CUMULATIVE DRIFT PROBLEM. THE ONE AFTER SHOWS HOW TO AVOID IT.

- OTHER TIMING PROBLEMS FOUND IN THIS TYPE OF SYSTEM ARE DESCRIBED IN "DESIGNING REAL TIME SYSTEMS IN Ada", FINAL REPORT, 1986 CHAPTERS 3 AND 4.

HOW DO WE DETERMINE THE LENGTH OF THE DELAYS?

- THE EXACT AMOUNT OF TIME NEEDED TO PERFORM AN ACTIVITY IS HARD TO DETERMINE.
  - THE AMOUNT OF PROCESSING MAY VARY.
  - PROCESSING MAY BE PREEMPTED BY ANOTHER TASK.
- HERE IS A FIRST APPROXIMATION:

```
loop -- 250 times a second (0.004 seconds per cycle)
   Start_Time := Clock; -- Function call on Calendar.Clock
   Activity_A;
   delay 0.004 - (Clock - Start_Time); -- Second Function call with
                                      -- different result
end loop;
```

- THERE IS STILL A PROBLEM, BECAUSE A DELAY STATEMENT DELAYS FOR <u>AT LEAST</u> THE SPECIFIED DURATION.
  - A TASK BECOMES <u>ELIGIBLE</u> FOR EXECUTION EXACTLY <u>t</u> SECONDS AFTER EXECUTING
    delay <u>t</u>;
    BUT THE RUNTIME SYSTEM MIGHT NOT ALLOCATE THE PROCESSOR TO THAT TASK IMMEDIATELY.
  - THIS CAN CAUSE <u>CUMULATIVE DRIFT</u>.

INSTRUCTOR NOTES

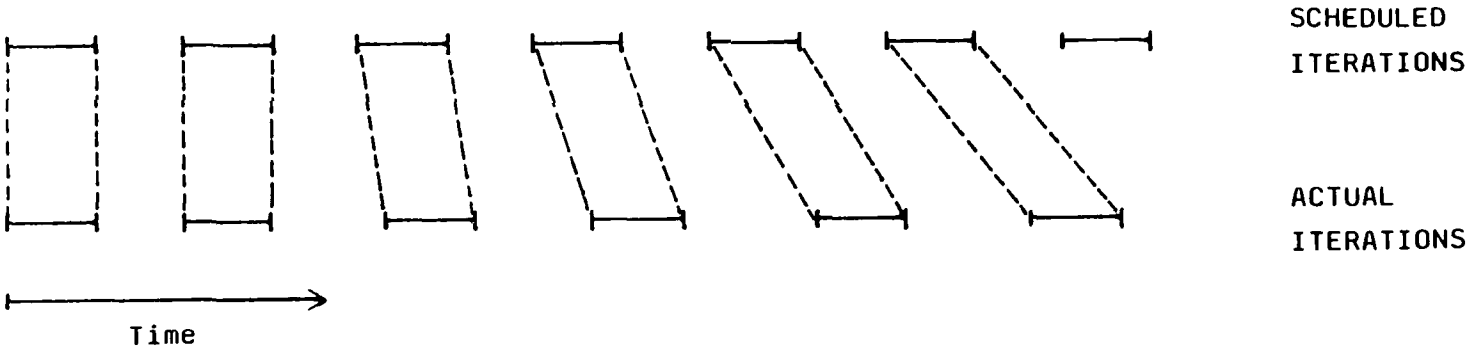
BULLETS 1 AND 2 DESCRIBE CUMULATIVE DRIFT. BULLETS 3 AND 4 DESCRIBE THE PROBLEMS IT CAUSES. THE NEXT SLIDE PRESENTS THE SOLUTION.

- BULLET 4:

  IN THE PICTURE IN BULLET 2, ONLY SIX ITERATIONS WERE ACTUALLY PERFORMED DURING THE PERIOD IN WHICH SEVEN ITERATIONS WERE SCHEDULED.

CUMULATIVE DRIFT

- A TASK SOMETIMES RESUMES EXECUTION A SHORT TIME AFTER ITS DELAY EXPIRES.
- LITTLE BY LITTLE, THE TASK FALLS FURTHER BEHIND.
  - IT CAN ONLY FALL BEHIND, NOT CATCH UP.
  - THE EFFECT IS CUMULATIVE



- THE LOOP GETS OUT OF PHASE WITH THE REST OF THE SYSTEM.
- IN THE LONG RUN, THE LOOP OPERATES AT LESS THAN THE REQUIRED FREQUENCY.

INSTRUCTOR NOTES

BOX, BULLET 2:

ASSUME Next_Iteration_Time IS THE TIME AT WHICH THE NEXT ITERATION IS SCHEDULED TO START. SINCE THE FUNCTION CALL CLOCK RETURNS THE CURRENT TIME, Next_Iteration_Time - CLOCK IS THE AMOUNT OF TIME FROM "NOW" UNTIL THE NEXT SCHEDULED ITERATION.

A_Task TASK BODY:

THE CONTEXT CLAUSE "with Calendar; use Calendar;" IS ASSUMED.

AS THE NEXT SLIDE EXPLAINS, THIS DOES NOT FORCE EACH ITERATION TO START ON TIME, BUT IT TENDS TO LIMIT THE LAG BETWEEN SCHEDULED AND ACTUAL START OF EACH ITERATION.

AVOIDING CUMULATIVE DRIFT

PROGRAMMING HINT

TO AVOID CUMULATIVE DRIFT:

- KEEP TRACK OF THE EXACT TIME THE LOOP IS SCHEDULED TO PERFORM THE NEXT ITERATION.
- SPECIFY A DELAY THAT EXPIRES AT EXACTLY THAT TIME:

```
delay Next_Iteration_Time - Clock;
```

```
task body A_Task is

   Cycle_Duration : constant := 0.004;
       -- Activity_A should be performed once every 4 milliseconds

   ...
begin
   ...
   Next_Iteration_Time := Clock;
   loop
       Activity_A;
       Next_Iteration_Time := Next_Iteration_Time + Cycle_Duration;
       delay Next_Iteration_Time - Clock;

   end loop;

end A_Task;
```

INSTRUCTOR NOTES

- BULLET 1: THIS IS FOR THE REASON THAT WAS DESCRIBED ON SLIDE 14-9.

- BULLET 2: THE LOOP IS SELF-CORRECTING. (EACH ITERATION CORRECTS FOR ANY TIME LAG IN THE PREVIOUS ITERATION BY SHORTENING ITS DELAY ACCORDINGLY.)

- BULLET 3: SCHEDULED LOOP ITERATION TIMES ARE COMPUTED INDEPENDENTLY OF ACTUAL ITERATION TIMES, BY THE STATEMENT

```
Next_Iteration_Time := Next_Iteration_Time + Cycle_Duration;
```

  THUS THE TASK BECOMES ELIGIBLE TO EXECUTE THE NEXT ITERATION EXACTLY ON SCHEDULE, EVEN IF THE PREVIOUS ITERATION FINISHED LATE.

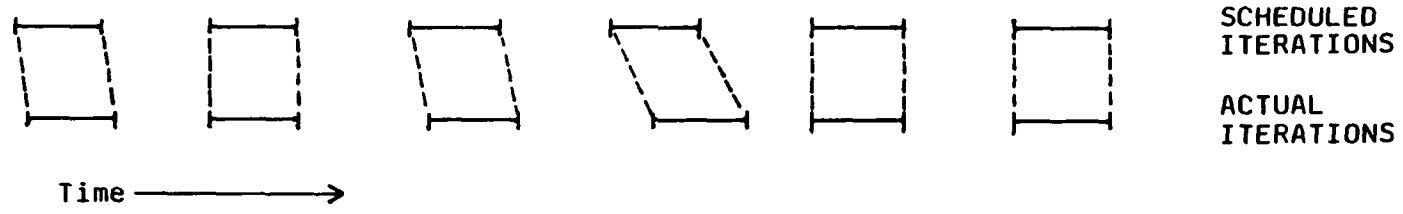
- BULLET 4: THIS SHOULD HAPPEN ONLY RARELY.

  - ITEM 1: DIAGRAM SHOWS THAT ITERATION DOES NOT BEGIN UNTIL LONG AFTER THE TASK BECOMES ELIGIBLE, BECAUSE THE CPU IS BUSY ELSEWHERE.
  - ITEM 2: A DELAY STATEMENT WITH A NEGATIVE VALUE ALWAYS SPECIFIES A NEGATIVE DELAY, SO IF ONE ITERATION FINISHES AFTER THE NEXT IS ALREADY DUE TO BEGIN, THE TASK IS ELIGIBLE TO START THE NEXT ITERATION IMMEDIATELY.
  - ITEM 3: WERE THE ITERATION EXECUTION TIME NOT CONSIDERABLY LESS THAN THE TIME BETWEEN CYCLES, IT WOULD BE IMPOSSIBLE TO COMPLETE PROCESSING OF THIS AND ALL OTHER TASKS EVEN UNDER NORMAL LOAD. THE LENGTH OF AN ITERATION RELATIVE TO THE TIME BETWEEN ITERATIONS IS EXAGGERATED IN THIS DIAGRAM TO MAKE THE PROCESSING OVERLOAD STAND OUT.

    NOTICE THAT THE OVERLOAD CONDITION IS QUICKLY CORRECTED AND THE TASK RESUMES NORMAL OPERATION WITHOUT SKIPPING AN ITERATION.

JITTER

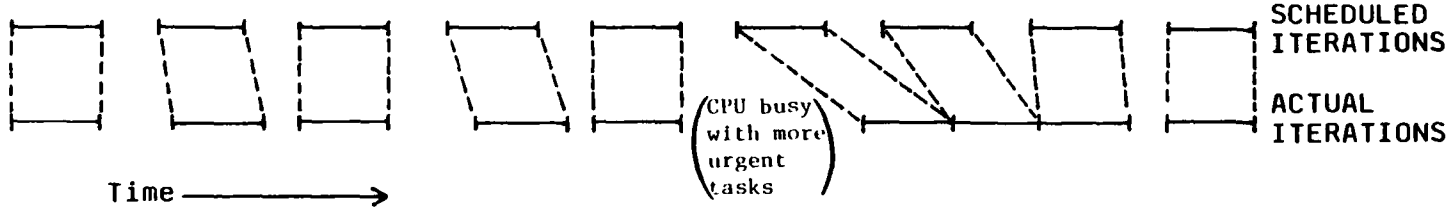
- THERE MAY STILL BE LAGS BETWEEN THE TIME A DELAY EXPIRES AND THE TIME A TASK RESUMES EXECUTION.
- THE LENGTH OF THE NEXT DELAY WILL BE SHORTENED ACCORDINGLY.
- USUAL CASE:
  - EACH DELAY EXPIRES PRECISELY WHEN THE NEXT ITERATION IS SCHEDULED.
  - TIME LAGS FOR DIFFERENT ITERATIONS ARE INDEPENDENT.



- TEMPORARY OVERLOAD:
  - THE TASK MAY FALL A FULL CYCLE OR MORE BEHIND.
  - THE EXPRESSION

    Next_Iteration_Time - Clock

    BECOMES NEGATIVE, SPECIFYING A ZERO-SECOND DELAY.
  - SINCE THE TIME TO EXECUTE ONE ITERATION SHOULD BE CONSIDERABLY LESS THAN THE TIME BETWEEN CYCLES, THE TASK WILL SOON CATCH UP:



- SMALL, INDEPENDENT DEVIATIONS FROM SCHEDULED START TIMES ARE KNOWN AS JITTER.

INSTRUCTOR NOTES

- BULLET 1:

  THE "PERFECT ADHERENCE" CURVE IS THE SET OF POINTS FOR WHICH ACTUAL START TIME = SCHEDULED START TIME.

- BULLET 2:

  THE CURVES FOR BULLET 2 ARE COMPUTED BY SUBTRACTING THE SOLID CURVES FOR BULLET 1 FROM THE DOTTED CURVES.

  THE KEY POINT IS THAT DRIFT IS CUMULATIVE BUT JITTER IS NOT. THE CURVE FOR JITTER EXPLAINS THE ORIGIN OF THAT TERM.

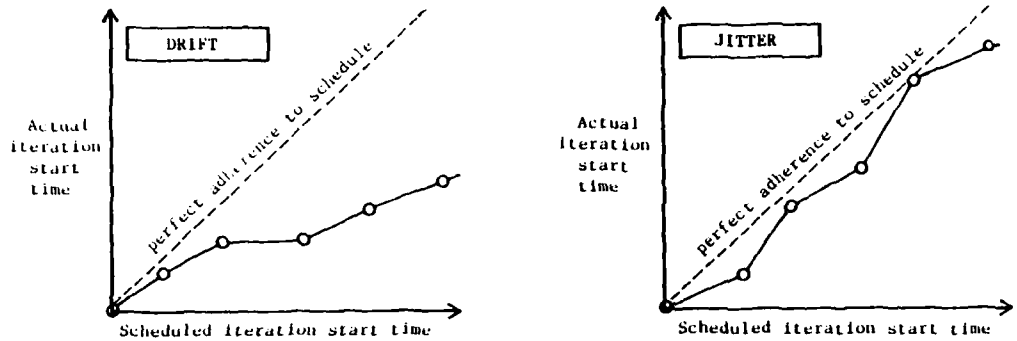
- BULLET 4:

  AN INTEGRATION COMPUTATION THAT ADJUSTS FOR JITTER WILL BE GIVEN ON SLIDE 14-15.

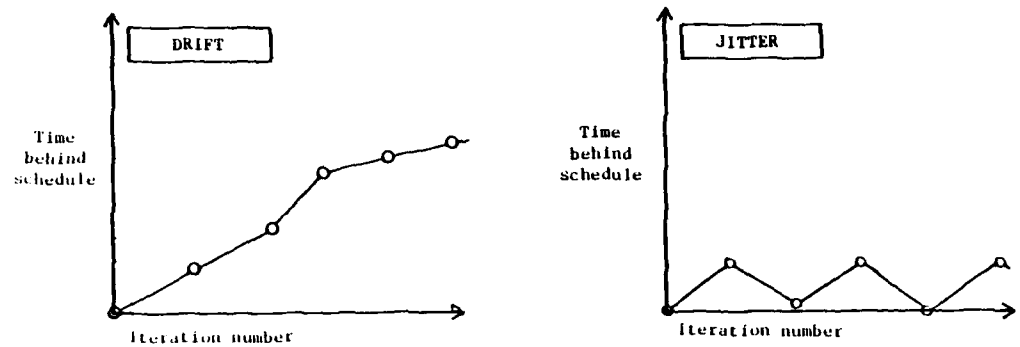
  - ITEM 3: THIS IS A JOKE.

JITTER VERSUS DRIFT

- ACTUAL vs. SCHEDULED START TIME:



- TIME BEHIND SCHEDULE:



- IN MOST CASES, A LIMITED AMOUNT OF JITTER IS ACCEPTABLE, BUT DRIFT IS NOT.
- "INTEGRATION" ALGORITHMS MUST BE ADJUSTED TO ACCOUNT FOR JITTER.
  - EXAMPLE: LOOP TO SAMPLE ACCELERATION AND COMPUTE VELOCITY.
  - INSTEAD OF ADDING A CONSTANT TIMES ACCELERATION TO PREVIOUS VELOCITY, MUST ADD ACTUAL TIME BETWEEN SAMPLES TIMES ACCELERATION.
  - FAILURE TO MAKE THIS ADJUSTMENT CAN RESULT IN JITTER BUGS.

INSTRUCTOR NOTES

- BULLET 1:

  WE ARE REFERRING TO DEPENDENCIES LIKE LEAVING THE HARDWARE IN A CERTAIN STATE. DATA DEPENDENCIES ARE COVERED ON THE NEXT SLIDE.

- BULLET 2:

  THE ENTRY CALL ACTS AS A GATE THAT KEEPS C_Task FROM BEGINNING TO EXECUTE Activity_C. BY ACCEPTING THE CALL ON WAIT, B_Task "OPENS THE GATE" TO LET C_Task THROUGH.

  THE "GATE" COULD ALSO HAVE BEEN IMPLEMENTED WITH THE ENTRY CALL GOING IN THE OTHER DIRECTION.

- BULLET 3:

  AS EXPLAINED ON PAGE 14-12i, THE TIME TO EXECUTE Activity_B SHOULD BE A SMALL FRACTION OF THE 13.3 ms CYCLE TIME. SIMILAR CONDITIONS SHOULD HOLD FOR OTHER CYCLIC TASKS. THIS IS WHAT IS MEANT BY "REASONABLE PROCESSOR LOAD."

- BULLET 4:

  THIS APPROACH CAN BE ADAPTED TO SYNCHRONIZE Activity_B AND Activity_C IN OTHER WAYS AS WELL, e.g. EACH ACTIVATION OF Activity_B FOLLOWED BY TWO ACTIVATIONS OF Activity_C. C_Task MUST BE REWRITTEN WITH NESTED LOOPS, WITH THE ENTRY CALL GOING IN THE OUTER LOOP.

- A CYCLIC EXECUTIVE MAY ALWAYS SCHEDULE Activity_B BEFORE Activity_C IN EACH MAJOR CYCLE BECAUSE OF TIMING DEPENDENCIES.
- WITH SINGLE-THREAD TASKS, TIMING OF Activity_C's LOOP CAN BE CONTROLLED BY A RENDEZVOUS INSTEAD OF A DELAY STATEMENT.

```
task B_Task is
   ...
   entry Wait;
end B_Task;

task body B_Task is
   Cycle_Duration:  constant := 0.0133;
   Next_Iteration_Time:  Time;
   ...
begin
   ...
   Next_Iteration_Time := Clock;
   loop
       Activity_B;
       accept Wait;
       Next_Iteration_Time :=
          Next_Iteration_Time + Cycle_Duration;
       delay Next_Iteration_Time - Clock;
   end loop;
end B_Task;
```

```
task C_Task is
   ...
end C_Task;

task body C_Task is
   ...
begin
   ...
   loop
      B_Task.Wait;
      Activity_C;
   end loop;
end C_Task;
```

- GIVEN REASONABLE PROCESSOR LOAD, C_Task WILL USUALLY BE WAITING AT THE ENTRY CALL BY THE TIME B_Task's DELAY EXPIRES AND B_Task GOES ON TO REACH THE ACCEPT STATEMENT.
- THERE IS ONE ITERATION OF B_Task's LOOP FOR EACH ITERATION OF C_Task's LOOP.

INSTRUCTOR NOTES

THIS IS A GENERALIZATION OF THE SYNCHRONIZATION APPROACH DESCRIBED ON THE PREVIOUS SLIDE.

Position_Task INTEGRATES SPEED TO OBTAIN POSITION. (THE INTEGRATION COMPUTATION MULTIPLIES THE AVERAGE OF TWO CONSECUTIVE SPEED READINGS BY THE TIME BETWEEN READINGS TO ESTIMATE THE DISTANCE TRAVELED BETWEEN THE TWO READINGS. BECAUSE OF JITTER, WE CANNOT SIMPLY TAKE THE TIME BETWEEN READINGS TO BE THE NOMINAL 0.1 SECOND CYCLE TIME.)

- BULLET 3:

  - ITEM 1: COMMUNICATION BY PARAMETERS IS PREFERABLE TO COMMUNICATION BY GLOBAL VARIABLES FOR THE SAME REASONS IN MULTITASK PROGRAMS AS IN SEQUENTIAL PROGRAMS.

COMMUNICATION AMONG ACTIVITIES

- A CYCLIC EXECUTIVE MAY ALWAYS SCHEDULE Activity_B BEFORE Activity_C IN EACH MAJOR CYCLE BECAUSE Activity_C USES DATA WHICH Activity_B PRODUCES.
- THE ENTRY USED TO SYNCHRONIZE ACTIVITIES CAN HAVE PARAMETERS FOR PASSING DATA.

```
task Position_Task is
   ...
   entry Get_Position (Position: out Position_Type);
end Position_Task;

task body Position_Task is
   Cycle_Duration: constant := 0.1;
   Next_Cycle_Time, Old_Reading_Time, New_Reading_Time:Time
   Old_Speed, New_Speed: Speed_Type;
   Current_Position: Position_Type;
   ...
begin
   ...
   loop
      New_Reading_Time := Clock;
      Read_Speed (New_Speed);
      Current_Position :=
         ((New_Speed + Old_Speed) / 2) *
         (New_Reading_Time - Old_Reading_Time);
      Old_Speed := New_Speed;
      Old_Reading_Time := New_Reading_Time;
      accept Get_Position (Position: out Position_Type) do
         Position := Current_Position;
      end Get_Position;
      Next_Cycle_Time := Next_Cycle_Time + Cycle_Duration;
      delay Next_Cycle_Time - Clock;
   end loop;
end Position_Task;
```

```
task Display_Task is
   ...
end Display_Task;

task body Display_Task is
   Position: Position_Type;
   ...
begin
   ...
   loop
      Position_Task.
       Get_Position
          (Position);
      ...
   end loop;
end Display_Task;
```

- NO NEED FOR SHARED GLOBAL DATA
  - EXPLICIT DATA FLOW, EXPLICIT INTERACTIONS
  - NO NEED TO WORRY ABOUT MUTUAL EXCLUSION

INSTRUCTOR NOTES

- BULLET 2:

  TO IMPLEMENT INTERLEAVED CONCURRENCY, THE RUNTIME SYSTEM IS CONSTANTLY MAKING DECISIONS ABOUT WHETHER TO SWITCH TO ANOTHER TASK AND WHICH TASK TO SWITCH TO. THESE DECISIONS, MADE OF RUNTIME, DETERMINE WHICH TASKS ARE ACTIVE WHEN. IN A PURE CYCLIC EXECUTIVE, THE RUNNING TIMES OF EACH TASK ARE CAREFULLY PLANNED DURING PROGRAM DESIGN.

- BULLET 3:

  - ITEM 1: SLIDES 14-11, 14-14, AND 14-15 EXPLAINED HOW TO USE DELAY STATEMENTS AND RENDEZVOUS TO ACHIEVE THIS EFFECT.
  - ITEM 2: THE NEXT SLIDE ADDRESSES THE PROBLEM OF ENSURING THAT THERE IS INDEED AMPLE CPU TIME AVAILABLE FOR ALL TASKS.
  - ITEM 3: WE RETURN TO THIS POINT AT THE END OF SECTION 20. AFTER DISCUSSING THE EXACT MEANING OF PRIORITIES. HIGHER PRIORITIES FOR HIGHER-FREQUENCY LOOPS ARE JUSTIFIED IN DETAIL AT THAT POINT.

SCHEDULING OF SINGLE-THREAD TASKS

- BASED ON DELAY STATEMENTS, ENTRY CALLS, SIMPLE ACCEPT STATEMENTS, AND SELECTIVE WAITS, CERTAIN TASKS ARE BLOCKED (INELIGIBLE FOR EXECUTION) AT ANY TIME.
- ON THE FLY, THE RUNTIME SYSTEM CHOOSES AN UNBLOCKED TASK FOR EXECUTION.
  - THIS CHOICE REPLACES THE SCHEDULING DONE AT DESIGN TIME FOR CYCLIC EXECUTIVES.
  - IT ENTAILS SOME RUNTIME OVERHEAD.
- CERTAIN CONSTRAINTS ENSURE THAT THE CHOICE RESULTS IN A DESIRABLE SCHEDULING OF TASKS:
  - PROPERLY DESIGNED DELAY STATEMENTS, ENTRY CALLS, AND ACCEPT STATEMENTS CAUSE TASKS TO BECOME UNBLOCKED AT THE REQUIRED FREQUENCY, BUT ONLY WHEN TASKS THAT MUST GO BEFORE IT HAVE EXECUTED.
  - IF THERE IS AMPLE CPU TIME FOR ALL THE REQUIRED PROCESSING, MOST TASKS WILL BE BLOCKED MOST OF THE TIME, SO EACH UNBLOCKED TASK WILL NORMALLY GET A CHANCE TO EXECUTE SHORTLY AFTER IT BECOMES UNBLOCKED.
  - TASKS CAN BE GIVEN PRIORITIES (DISCUSSED IN SECTION 20).
    - AN UNBLOCKED TASK OF HIGHER PRIORITY IS ALWAYS SELECTED FOR EXECUTION BEFORE ONE OF LOWER PRIORITY.
    - HIGHER-FREQUENCY LOOPS CAN BE GIVEN HIGHER PRIORITY.
    - THIS WILL TEND TO REDUCE JITTER. LOWER-FREQUENCY TASKS HAVE A GREATER OPPORTUNITY TO BE SCHEDULED WITHOUT FALLING BEHIND.

INSTRUCTOR NOTES

- BULLET 2:

  - ITEM 4: $\underline{p}$ IS THE AVERAGE PERIOD BETWEEN ITERATIONS, SUBJECT TO JITTER

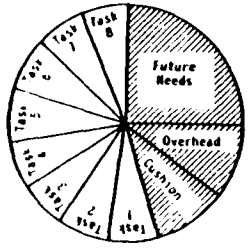
  - ITEM 6: SECTION 25 OF THE COURSE DEALS WITH WAYS OF REDUCING THE VALUE OF $\underline{c}$ (ACCOMPLISHING THE SAME PROCESSING WITH FEWER INSTRUCTIONS).

- A MORE DETAILED COMPARISON OF THESE TECHNIQUES IS PROVIDED BY "DESIGNING REAL TIME SYSTEMS IN Ada", FINAL REPORT, 1986 CHAPTERS 2 THROUGH 6.

BUDGETING TIME

- PURE CYCLIC EXECUTIVE:
  - TOTAL RUNNING TIME OF ALL ACTIVITIES SCHEDULED FOR A GIVEN MINOR CYCLE MUST BE LESS THAN THE MINOR CYCLE LENGTH.
  - SIMPLE CRITERION, BUT ACHIEVING IT IS DIFFICULT.

- SINGLE-THREAD TASKS:
  - ALLOCATE A PORTION OF THE PROCESSOR TIME TO EACH TASK.
  - LEAVE A PORTION UNALLOCATED FOR:
    - RUNTIME SYSTEM OVERHEAD (AMOUNT BASED ON EXPERIENCE)
    - FUTURE ENHANCEMENTS
    - CUSHION FOR TEMPORARY PROCESSING OVERLOAD
  - LET c BE THE ESTIMATED MAXIMUM CPU TIME FOR ONE ITERATION OF THE TASK'S MAIN LOOP.
    - DOES NOT INCLUDE TIME SPENT IN DELAYS
    - DOES NOT INCLUDE TIME WAITING FOR RENDEZVOUS
  - LET p BE THE NOMINAL PERIOD OF THE LOOP (TIME BETWEEN ITERATIONS)
  - THE TASK MUST BE RUNNING c/p OF THE TIME TO DO THE PROCESSING REQUIRED FOR ONE ITERATION.
  - c/p MUST NOT BE GREATER THAN THE TASK'S ALLOCATED PORTION OF THE PROCESSOR TIME.
  - MORE COMPLICATED CRITERION, BUT RUNTIME SYSTEM IS RESPONSIBLE FOR ACHIEVING IT.




```
loop
   [computation]
   Next := Next + p;
   delay Next-Clock;
end loop;
```

c SECONDS CPU TIME
p SECONDS REAL TIME

INSTRUCTOR NOTES

AS MACLAREN POINTED OUT, THE CYCLIC EXECUTIVE APPROACH IS REALLY ONLY ADVANTAGEOUS FOR VERY SIMPLE, COMPLETELY SYNCHRONOUS PROBLEMS.

BOTH APPROACHES TO CYCLIC PROCESSING ARE AVAILABLE WITHIN Ada.

PURE CYCLIC EXECUTIVE VERSUS SINGLE-THREAD TASKS

ADVANTAGES OF PURE CYCLIC EXECUTIVE

- FAMILIARITY
- SIMPLICITY OF RUNTIME SYSTEM
- EFFICIENCY (LOW OVERHEAD)
- PREDICTABILITY
  - ACTIONS ALWAYS PERFORMED IN THE SAME ORDER
  - BEHAVIOR IS REPRODUCIBLE

ADVANTAGES OF SINGLE-THREAD TASKS

- MORE NATURAL PARTITION OF PROBLEM
  - ONE TASK PER REAL-WORLD ACTIVITY
  - WELL-DEFINED INTERFACES AND DATA FLOW
- FLEXIBILITY
  - SCHEDULING OF THE PROCESSOR CAN CHANGE DYNAMICALLY TO MEET VARYING LOADS
    - URGENT TASKS CAN "OVERFLOW" WHEN NECESSARY, ALLOWING LESS URGENT TASKS TO CATCH UP LATER.
    - NO PREIMPOSED IDLE PERIODS WHEN THERE IS WORK TO BE DONE.
  - CYCLE LENGTHS CAN BE PRECISELY FITTED TO NATURAL PROBLEM REQUIREMENTS
    - NEED NOT BE MINOR CYCLE LENGTH TIMES A POWER OF TWO
    - BY NOT CYCLING MORE FREQUENTLY THAN NECESSARY, WE CONSERVE PROCESSING POWER.
  - EVENT-DRIVEN ASYNCHRONOUS PROCESSING EASILY ACCOMMODATED.
- EASE OF DESIGN
  - NO CYCLE ALLOCATION PROBLEM
  - PROCESSOR SCHEDULING HANDLED BY RUNTIME SYSTEM

INSTRUCTOR NOTES

ALLOW 90 MINUTES FOR THIS SECTION, NOT INCLUDING THE EXERCISE ON SLIDE 15-23.

SLIDES 15-1 THROUGH 15-22 SHOULD BRING YOU TO THE END OF DAY 3. PRESENT THE EXERCISE ON SLIDE 15-23 AT THE BEGINNING OF DAY 4, AND ALLOW 90 MINUTES FOR IT.

Section 15

STREAM-ORIENTED TASK DESIGN

INSTRUCTOR NOTES

BEFORE EXPLAINING STREAM-ORIENTED TASK DESIGN, GIVE STUDENTS A CHANCE TO WRESTLE WITH THE KIND OF PROBLEM THIS APPROACH HELPS TO SOLVE.

THIS SLIDE CONTAINS A SHORT EXERCISE FOR STUDENTS TO WORK ON INDEPENDENTLY. TELL STUDENTS NOT TO LOOK AHEAD IN THEIR NOTES. TELL THEM TO SOLVE THE PROBLEM WITHOUT USING TASKS EVEN IF THEY HAVE AN IDEA ABOUT HOW TASKS MIGHT BE HELPFUL.

THE PROCEDURE Get_Next_Block READS A BLOCK FROM THE SPECIFIED CHANNEL, SETS Block_Size TO THE NUMBER OF CHARACTERS IN THE BLOCK, AND FILLS Into_Buffer (1 .. Block_Size) WITH THOSE CHARACTERS. THE REMAINING COMPONENTS OF Into_Buffer ARE LEFT UNDEFINED.

GIVE THE CLASS ABOUT 15 MINUTES TO START TO FORMULATE A SOLUTION. THEN INTERRUPT THEM EVEN THOUGH MOST STUDENTS WILL NOT HAVE FINISHED. EXPLAIN THAT THE PURPOSE OF THE EXERCISE WAS ONLY TO GIVE THEM A FEEL FOR THE COMPLEXITY OF THE PROBLEM, NOT TO FORMULATE A COMPLETE SOLUTION. HAVE A FEW STUDENTS DESCRIBE WHAT WAS DIFFICULT ABOUT THE PROBLEM AND PROMISE THAT A COMPLETE SOLUTION WILL BE GIVEN AT THE END OF THIS SECTION.

A SEQUENTIAL PROBLEM

- WRITE A FUNCTION THAT COMPARES TWO MESSAGES BEING RECEIVED ON CHANNEL 1 AND CHANNEL 2 AND RETURNS A Boolean VALUE INDICATING WHETHER THE MESSAGES ARE THE SAME. (ASSUME THAT THIS IS THE ONLY WAY IN WHICH THE MESSAGES WILL BE USED.)

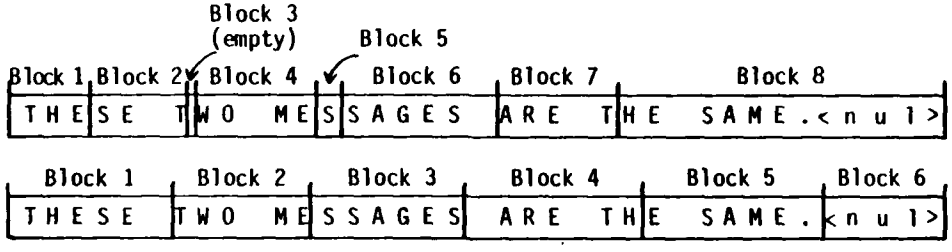
- MESSAGES ARE BROKEN INTO BLOCKS OF UP TO 128 CHARACTERS EACH. A CHANNEL CAN BE READ BLOCK-BY-BLOCK USING THE FOLLOWING PACKAGE:

```
package Block_Package is
   type Channel_Number_Type is range 1 .. 2;
   subtype Block_Buffer_Subtype is String (1 .. 128);
   subtype Character_Count_Subtype is Integer range 0 .. 128;
   procedure Get_Next_Block
      (From_Channel : in Channel_Number_Type;
       Into_Buffer  : out Block_Buffer_Subtype;
       Block_Size   : out Character_Count_Subtype);
end Block_Package;
```

  BLOCK SIZES ARE ARBITRARY AND MESSAGES MAY INCLUDE EMPTY BLOCKS.

- A NULL BYTE (ASCII.NUL) OCCURS AT THE END OF A MESSAGE AND NOWHERE ELSE. SUBSEQUENT CALLS ON Get_Next_Block WILL FETCH EMPTY BLOCKS.

- MESSAGES WITH THE SAME CONTENTS SHOULD BE CONSIDERED THE SAME, REGARDLESS OF HOW THEY ARE BROKEN INTO BLOCKS:



INSTRUCTOR NOTES

- BULLET 2:

  A TRANSFORMATION WITH ZERO INPUT STREAMS COULD BE ONE PRODUCING PULSES EVERY 100 MILLISECONDS OR ONE PRODUCING A STREAM OF Boolean VALUES, WHERE THE nth VALUE IN THE STREAM IS TRUE IF AND ONLY IF n mod 3 = 0.

- BULLET 4:

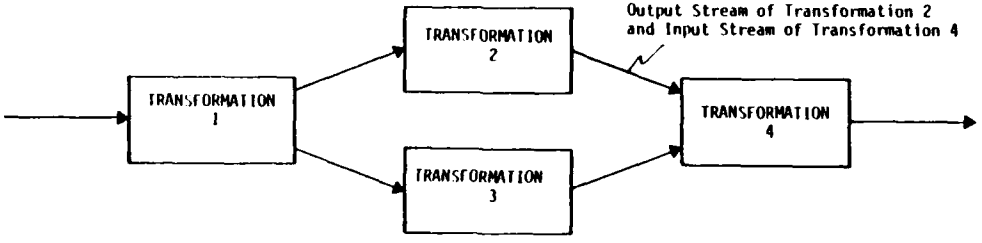
  THE SECOND AND THIRD SUBBULLETS EXPLAIN WHY STREAM-ORIENTED TASK DESIGN REDUCES COMPLEXITY. IT DECOMPOSES A COMPLEX PROBLEM INTO SEVERAL INDEPENDENT SIMPLE SUBPROBLEMS. IT ALSO SEPARATES CONCERN WITH THE DECOMPOSITION FROM CONCERN WITH THE SOLUTIONS TO THE SUBPROBLEMS.

STREAM-ORIENTED TASK DESIGN

- SOME PROBLEMS ARE MOST EASILY UNDERSTOOD AS A COLLECTION OF TRANSFORMATIONS ON STREAMS OF DATA.
- EACH TRANSFORMATION HAS ZERO OR MORE INPUT STREAMS AND ONE OR MORE OUTPUT STREAMS.



- EACH TRANSFORMATION IS SIMPLE
  - OBTAIN DATA ITEMS IN ORDER FROM THE INPUT STREAMS
  - SEND DATA ITEMS IN ORDER TO THE OUTPUT STREAMS
  - STRAIGHTFORWARD WAY TO DERIVE OUTPUT ITEMS FROM INPUT ITEMS
- BY CONNECTING SIMPLE TRANSFORMATIONS WE PRODUCE MORE COMPLEX TRANSFORMATIONS
  - ONE TRANSFORMATION'S OUTPUT STREAM SERVES AS ANOTHER TRANSFORMATION'S INPUT STREAM.



  - INTERCONNECTION TREATS TRANSFORMATIONS AS "BLACK BOXES."
  - TRANSFORMATIONS WRITTEN WITHOUT CONCERN FOR THE INTERCONNECTIONS.

INSTRUCTOR NOTES

- BULLET #1:

  JSP AND JSD ARE SOFTWARE ENGINEERING METHODOLOGIES BASED ON DATA STREAMS AND TRANSFORMATIONS.

  REFER STUDENTS TO THE JACKSON REFERENCE IN THE BIBLIOGRAPHY.

  AS WE SHALL SEE LATER, IT IS EASY TO IMPLEMENT STREAM-ORIENTED DESIGNS WITH ADA TASKS. THIS MAKES THE JACKSON METHODOLOGY EXTREMELY SUITABLE FOR USE WITH ADA.

- BULLET 2:

  BECAUSE THE FILES ARE SEQUENTIAL, EACH FILE ELEMENT IS A LINE. THUS FILES MUST BE READ AND WRITTEN LINE-BY-LINE.

  THE NEXT FEW SLIDES PURSUE THIS PROBLEM IN GREATER DETAIL.

  ANOTHER EXAMPLE OF A STRUCTURE CLASH IS IN THE MESSAGE COMPARISON PROBLEM. IN THIS CASE THE STRUCTURES OF THE INPUT STREAMS CLASH BECAUSE MESSAGES CAN BE BROKEN INTO BLOCKS IN DIFFERENT WAYS IN EACH STREAM.

  IT IS FOR PROBLEMS WITH STRUCTURE CLASHES THAT STREAM-ORIENTED DESIGN IS MOST USEFUL.

- BULLET 3:

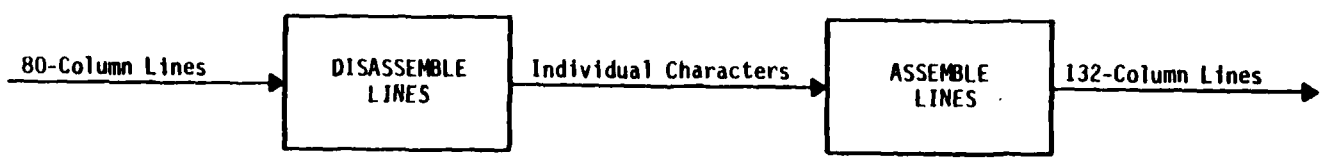
  THE REFORMATTING IS BROKEN INTO TWO VERY SIMPLE TRANSFORMATIONS.

  THE FIRST TAKES AN INPUT STREAM OF 80-COLUMN LINES AND PRODUCES AN OUTPUT STREAM CONSISTING OF THE INDIVIDUAL CHARACTERS IN THOSE LINES.

  THE SECOND TAKES AN INPUT STREAM OF INDIVIDUAL CHARACTERS, GROUPS THEM INTO 132-CHARACTER LINES, AND PRINTS THE LINES.

STRUCTURE CLASHES

- JACKSON STRUCTURED PROGRAMMING (JSP) AND JACKSON SYSTEM DEVELOPMENT (JSD) BASE PROGRAM STRUCTURE ON THE STRUCTURE OF INPUT AND OUTPUT STREAMS.
  - PROGRAMS EASY TO WRITE WHEN STRUCTURES MATCH
  - PROGRAMS HARD TO WRITE WHEN STRUCTURES CLASH
- EXAMPLE OF A STRUCTURE CLASH:
  - PROGRAM TO REFORMAT A SEQUENTIAL FILE OF 80-COLUMN LINES INTO A FILE OF 132-CHARACTER LINES, CHARACTER-FOR-CHARACTER.
  - INPUT STRUCTURE: 80-COLUMN LINES
  - OUTPUT STRUCTURE: 132-CHARACTER LINES
- STREAM TRANSFORMATIONS CAN BE USED TO RESOLVE STRUCTURE CLASHES:



INSTRUCTOR NOTES

- BULLET 2:

  A "NULL BYTE" IS THE Character VALUE ASCII.NUL. (THE PACKAGE ASCII PROVIDES MNEMONIC NAMES FOR CONTROL CHARACTERS AND CERTAIN OTHER CHARACTERS. IT IS DECLARED INSIDE THE PACKAGE Standard, SO IT SHOULD NOT BE USED IN A with CLAUSE.)

- BULLET 3:

  THE MAIN LOOP REPEATEDLY READS INPUT LINES INTO THE INPUT BUFFER, THEN EXECUTES A for LOOP TO DISPOSE OF EACH CHARACTER IN THE BUFFER. DISPOSING OF A CHARACTER CONSISTS OF PLACING IT IN THE NEXT POSITION IN THE OUTPUT BUFFER (INDEXED BY Output_Position), WRITING THE LINE AND RESETTING THE OUTPUT POSITION IF THE BUFFER IS FULL, AND ADVANCING THE OUTPUT POSITION OTHERWISE. ONCE A NULL BYTE IS ENCOUNTERED, WE EXIT THE MAIN LOOP AND EXECUTE THE CLOSING for LOOP, WHICH FILLS THE REMAINING PLACES IN THE OUTPUT BUFFER (POSSIBLY ALL 132 PLACES) WITH NULL BYTES. WE THEN WRITE THE CLOSING LINE AND FINISH.

  THE SOLUTION IS SURPRISINGLY COMPLEX FOR SUCH A SIMPLE PROBLEM. THIS IS BECAUSE THE SAME PART OF THE PROGRAM MUST KEEP TRACK OF BOTH THE CURRENT POSITION IN THE INPUT BUFFER AND THE CURRENT POSITION IN THE OUTPUT BUFFER. THE STREAM-ORIENTED SOLUTION ON THE NEXT SLIDE SEPARATE THESE CONCERNS.

  THE DIFFERENCE IN COMPLEXITY BETWEEN THE TWO SOLUTIONS IS NOT NEARLY AS STRIKING AS IT WILL BE LATER WHEN WE ADD A SMALL COMPLICATION TO THE PROBLEM.

A CLOSER LOOK AT THE LINE REFORMATTING PROBLEM

- BASIC OPERATIONS ARE READING AN 80-CHARACTER LINE AND WRITING A 132-CHARACTER LINE.
- END OF FILE:
  - LAST LINE OF INPUT FILLED WITH NULL BYTES AFTER LAST CHARACTER OF DATA. (IF LAST CHARACTER OF DATA IS IN COLUMN 80, IT IS FOLLOWED BY A LINE OF NULL BYTES.)
  - NO OTHER NULL BYTES OCCUR IN THE INPUT FILE.
  - LAST LINE OF OUTPUT TO BE MARKED IN THE SAME WAY.
- A TRADITIONAL SOLUTION:

```
with Read_Line, Write_Line;
procedure Reformat is
   Input_Buffer    : String (1 .. 80);
   Output_Buffer   : String (1 .. 132);
   Output_Position : Integer range Output_Buffer'Range := 1;
   Next_Character  : Character;
begin
   Main_Loop:
      loop
         Read_Line (Input_Buffer);
         for I in Input_Buffer'Range loop
            Next_Character := Input_Buffer (I);
            exit Main_Loop when Next_Character = ASCII.NUL;
            Output_Buffer (Output_Position) := Next_Character;
            if Output_Position = Output_Buffer'Last then
               Write_Line (Output_Buffer);
               Output_Position := 1;
            else
               Output_Position := Output_Position + 1;
            end if;
         end loop;
      end loop Main_Loop;
   for I in Output_Position .. Output_Buffer'Last loop
      Output_Buffer (I) := ASCII.NUL;
   end loop;
   Write_Line (Output_Buffer);
end Reformat;
```

INSTRUCTOR NOTES

AS IN MODULE L305, WE USE BOXES TO SURROUND ABSTRACT OPERATIONS WHOSE IMPLEMENTATIONS WILL BE GIVEN LATER. SLIDES 15-7 THROUGH 15-9 ADDRESS THE IMPLEMENTATION OF THE SEND-TO-STREAM AND RECEIVE-FROM-STREAM OPERATIONS IN Ada.

- BULLET 1:

  THIS TRANSFORMATION IS CONCERNED ONLY WITH TAKING CHARACTERS OUT OF THE INPUT BUFFER. NOW "DISPOSING OF" A CHARACTER SIMPLY MEANS SENDING IT. A SINGLE NULL BYTE ENDS THE OUTPUT STREAM.

- BULLET 2:

  THIS TRANSFORMATION IS CONCERNED ONLY WITH FILLING AND PRINTING OUTPUT BUFFERS. BECAUSE THE ORIGINAL COLUMN IN WHICH A CHARACTER WAS FOUND IS IRRELEVANT HERE, THIS TRANSFORMATION CAN BE DESIGNED AROUND THE STRUCTURE OF THE OUTPUT BUFFER. THE TRANSFORMATION REPEATEDLY EXECUTES A for LOOP TO FILL THE BUFFER WITH 132-CHARACTERS FROM THE INPUT STREAM AND THEN PRINTS THE BUFFER. THERE IS NO NEED TO EXPLICITLY CHECK THE CURRENT OUTPUT POSITION. (THE for LOOP DOES THIS IMPLICITLY.)

  ANY APPEARANCE OF COMPLEXITY IN THE for LOOP ARISES FROM THE NECESSITY TO SAVE THE VALUE OF THE LOOP PARAMETER BEFORE EXITING. CONCEPTUALLY (WITH THE ENTIRE if STATEMENT VIEWED AS A MODIFIED exit OPERATION), THE LOOP IS EXTREMELY SIMPLE.

THE STREAM-ORIENTED SOLUTION

- THE "DISASSEMBLE LINES" TRANSFORMATION:

```
Main_Loop:
   loop
      Read_Line (Input_Buffer);
      for I in Input_Buffer'Range loop
         send Input_Buffer (I) to the output stream
         exit Main_Loop when Input_Buffer (I) = ASCII.NUL;
      end loop;
   end loop Main_Loop;
```

- THE "ASSEMBLE LINES" TRANSFORMATION:

```
Main_Loop:
   loop
      for I in Output_Buffer'Range loop
         receive Output_Buffer (I) from input stream ;
         if Output_Buffer (I) = ASCII.NUL then
            J := I;
            exit Main_Loop;
         end if;
      end loop;
      Write_Line (Output_Buffer);
   end loop Main_Loop;
   for I in J + 1 .. Output_Buffer'Last loop
      Output_Buffer (I) := ASCII.NUL;
   end loop;
   Write_Line (Output_Buffer);
```

INSTRUCTOR NOTES

THE NEXT GENERATION OF PROGRAMMING LANGUAGES AND MACHINES MAY BE BASED ON THE DATA FLOW MODEL. WRITING DATA FLOW Ada PROGRAMS MAY MAKE THESE PROGRAMS MORE ADAPTABLE IN THE FUTURE.

- BULLET 2: IN A DATA FLOW COMPUTATION, ACTIONS ARE NOT ORDERED BY AN INSTRUCTION COUNTER, BUT BY THE AVAILABILITY OF DATA. A TRANSFORMATION "FIRES" WHEN ITS INPUTS ARE AVAILABLE, AND GENERATES AN OUTPUT. THIS OUTPUT MAY IN TURN CAUSE ANOTHER TRANSFORMATION TO FIRE.

  IN THE TEXT REFORMATTING PROBLEM, WE HAVE STARTED WITH FAIRLY HIGH-LEVEL TRANSFORMATIONS, IMPLEMENTED USING VON NEUMAN-STYLE Ada. IN A PURE DATA FLOW LANGUAGE, EVEN ASSEMBLY AND DISASSEMBLY OF LINES WOULD BE SPECIFIED IN TERMS OF MORE PRIMITIVE TRANSFORMATIONS.

- BULLETS 3 AND 4: IF STUDENTS ARE INTERESTED IN DETAILS, REFER THEM TO THE REFERENCE IN BULLET 6, PARTICULARLY THE INTRODUCTORY ARTICLE.

- BULLET 5: ACOS (ASP COMMON OPERATIONAL SOFTWARE) IS A DATA FLOW METHODOLOGY FOR PROGRAMMING THE NAVY'S AN/UYS-1 ADVANCED SIGNAL PROCESSOR (ASP). ECOS (EMSP COMMON OPERATIONAL SOFTWARE) IS A SIMILAR METHODOLOGY FOR PROGRAMMING THE NAVY'S ENHANCED MODULAR SIGNAL PROCESSOR (EMSP).

  THE IMPORTANT POINT TO BE MADE HERE IS THAT DATA FLOW PROGRAMMING IS NOT A WILD ACADEMIC PIPE DREAM. IT IS APPLICABLE TO EMBEDDED MILITARY APPLICATIONS.

- BULLET 6: THIS REFERENCE IS IN THE BIBLIOGRAPHY.

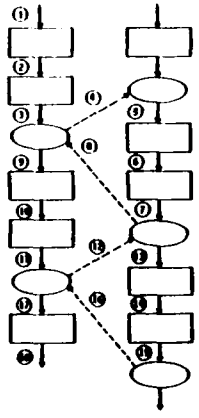
DATA FLOW PROGRAMMING

- DATA FLOW PROGRAMMING IS A STYLE OF PROGRAMMING BASED ON DATA STREAMS AND TRANSFORMATIONS.
- IT IS BASED ON A DIFFERENT VIEW OF THE NATURE OF COMPUTATION.
  - THE TRADITIONAL, OR VON NEUMANN MODEL:
    - GLOBAL, ADDRESSABLE MEMORY THAT IS FREQUENTLY UPDATED.
    - INSTRUCTIONS EXECUTED IN FIXED SEQUENCE, USING AN INSTRUCTION COUNTER.
  - THE DATA FLOW MODEL:
    - NO NAMED CONTAINERS (VARIABLES), ONLY VALUES
    - TRANSFORMATIONS ARE SIMPLE OPERATIONS, E.G. +, -, *, /
    - WHEN AN INPUT VALUE IS AVAILABLE FROM EACH INPUT STREAM, THE TRANSFORMATION COMPUTES A RESULT VALUE AND PLACES IT IN THE OUTPUT STREAM.
    - DATA FLOW "PROGRAMS" ARE JUST STREAM/TRANSFORMATION DIAGRAMS.
- DATA FLOW COMPUTERS HAVE BEEN DESIGNED AND BUILT.
- PROGRAMMING LANGUAGES HAVE BEEN DESIGNED AROUND THE DATA FLOW MODEL OF COMPUTATION.
- DATA FLOW METHODS HAVE BEEN APPLIED TO EMBEDDED SIGNAL-PROCESSING APPLICATIONS (ACOS, ECOS).
- FEBRUARY 1982 ISSUE OF IEEE COMPUTER IS DEVOTED TO DATA FLOW SYSTEMS.

INSTRUCTOR NOTES

THE MAIN POINT OF THIS SLIDE IS THAT STREAM-ORIENTED PROGRAMMING PREDATES Ada. Ada SIMPLY PROVIDES US WITH A MORE ADVANCED MECHANISM WITH WHICH STREAM-ORIENTED PROGRAMS CAN BE MORE CONVENIENTLY WRITTEN.

- BULLET 1: THE FIGURE PROVIDES THE CONTEXT FOR THE EXAMPLES THAT FOLLOW ON THIS SLIDE AND THE NEXT.
- BULLET 2:
  - ITEM 1: MULTIPLE PASSES ARE OBVIOUSLY NOT A REAL-TIME SOLUTION
  - ITEM 2: TRACE THE FLOW OF CONTROL ON THE SLIDE:

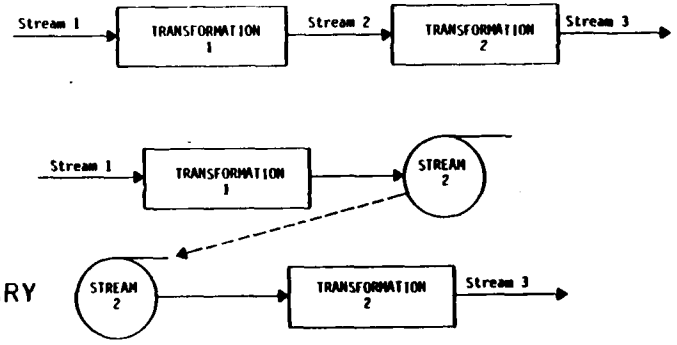


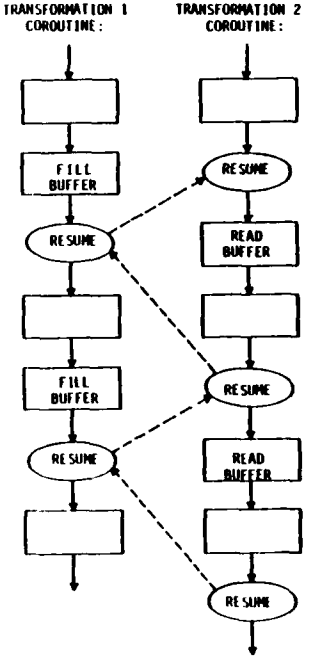
THE NEXT SLIDE DESCRIBES VARIATIONS ON COROUTINES.

- SUBITEM 2: SUBPROGRAMS HAVE A HIERARCHICAL RELATIONSHIP: ONE CALLS THE OTHER. COROUTINES HAVE A SYMMETRIC RELATIONSHIP. EACH CAN VIEW A RESUME OPERATION AS A CALL ON THE OTHER COROUTINE, AND THE OTHER COROUTINES RESUME AS A RETURN.
- SUBITEM 3: THE COROUTINES ARE NEVER ACTIVE AT THE SAME TIME.
- SUBITEMS 5 AND 6: IN THIS CONTEXT WE ARE USING THE WORD BUFFER TO MEAN A VARIABLE CAPABLE OF HOLDING ONE DATA ITEM.

IMPLEMENTATION OF STREAM OPERATIONS

- OPERATIONS ARE:
  - SENDING ONE ITEM TO A STREAM
  - RECEIVING ONE ITEM FROM A STREAM
- POSSIBLE IMPLEMENTATIONS (WHEN NOT USING ADA):
  - MULTIPLE PASSES
    - ONE PASS FOR EACH TRANSFORMATION
    - ENTIRE STREAM STORED IN A TEMPORARY FILE
  - COROUTINES
    - SEQUENCES OF INSTRUCTIONS THAT WORK *COOPERATIVELY*.
    - INSTEAD OF ONE COROUTINE CALLING A SECOND AND THE SECOND ONE RETURNING (AS WITH SUBROUTINES), COROUTINES RESUME EACH OTHER.
    - RESUMING ENTAILS SAVING OWN NEXT INSTRUCTION ADDRESS AND BRANCHING TO SAVED NEXT INSTRUCTION ADDRESS OF OTHER COROUTINE.
    - ONE COROUTINE FOR EACH TRANSFORMATION
    - SEND DATA TO OUTPUT STREAM BY PLACING AN ITEM IN A BUFFER AND RESUMING THE CONSUMING COROUTINE.
    - RECEIVE DATA FROM INPUT STREAM BY RESUMING THE PRODUCING COROUTINE AND THEN EXAMINING THE BUFFER.





INSTRUCTOR NOTES

THIS SLIDE RELATES COROUTINES TO SIMILAR IDEAS THAT STUDENTS MAY ENCOUNTER OR MAY ALREADY BE FAMILIAR WITH.

- BULLET 1:
  - ITEM 3: TRANSFORMATION_1 < Stream_1 MEANS INVOKE THE FILTER.
    TRANSFORMATION_1, WITH FILE Stream_1 USED AS THE SOURCE OF INPUT.
    TRANSFORMATION_2 > Stream_3 MEANS INVOKE THE FILTER.
    TRANSFORMATION_2, WITH FILE Stream_3 USED AS THE DESTINATION OF OUTPUT.
    THE VERTICAL BAR MEANS THAT THE OUTPUT OF Transformation_1 IS PIPED TO THE INPUT OF Transformation_2.
    (THE NAMES REFER TO THE EXAMPLE ON THE PREVIOUS SLIDE).

- BULLET 2:
  PROGRAM INVERSION IS USEFUL IN Ada AS AN OPTIMIZATION TOOL. IT IS DISCUSSED IN SECTION 24.

VARIATIONS ON COROUTINES

- "PIPES" AND "FILTERS":
  - ONE-INPUT-STREAM/ONE-OUTPUT-STREAM TRANSFORMATIONS WRITTEN AS SEPARATE PROGRAMS, CALLED "FILTERS."
  - FILTERS RECEIVE DATA FROM THE INPUT STREAM BY READING FROM THE STANDARD INPUT FILE AND SEND DATA TO THE OUTPUT STREAM BY WRITING TO THE STANDARD OUTPUT FILE. (ORDINARY I/O.)
  - THE OPERATING SYSTEM COMMAND LANGUAGE ALLOWS FILTERS TO BE CONNECTED BY "PIPES." DATA WRITTEN BY ONE FILTER WILL AUTOMATICALLY BE "PIPED TO" THE OTHER FILTER AND USED AS INPUT DATA.

    Transformation_1 < Stream_1 | Transformation_2 > Stream_3
  - PIPED TRANSFORMATIONS ARE AUTOMATICALLY RUN AS COROUTINES.
  - ORIGINATED WITH UNIX*.
  - ALLOWED SYSTEM TO PROVIDE VERY BASIC COMMANDS THAT COULD EASILY BE COMBINED TO DO COMPLEX JOBS.
- PROGRAM INVERSION (JACKSON):
  - PROGRAM ORIGINALLY WRITTEN IN HIGH-LEVEL LANGUAGE, BUT WITH send AND receive "PSEUDO-OPERTIONS" ON STREAMS.
  - INVERSION SYSTEMATICALLY TRANSFORMS THE PROGRAM TEXT SO THAT THE send AND receive OPERATIONS ARE REALIZED BY SIMULATING COROUTINES.
  - THE TRANSFORMATION CAN BE DONE AUTOMATICALLY.

*UNIX IS A TRADEMARK OF BELL LABS.

INSTRUCTOR NOTES

- BULLET 1:

  COROUTINES CAN BE VIEWED AS A RESTRICTED FORM OF CONCURRENCY, IN WHICH INTERLEAVING IS RIGIDLY CONTROLLED.

- BULLET 2:

  THE RENDEZVOUS CAN ALSO WORK IN THE OTHER DIRECTION, WITH THE SENDING TASK ACCEPTING A "Get_Data" ENTRY CALL FROM THE RECEIVING TASK AND PLACING THE DATA IN AN out PARAMETER.

  IN FACT, THAT IS THE APPROACH TAKEN AT THE END OF THIS SECTION TO SOLVE THE MESSAGE COMPARISON PROBLEM.

- BULLET 4:

  IF THIS APPROACH IS TAKEN, THE RENDEZVOUS IS NOT REVERSIBLE.

STREAM OPERATIONS IN Ada

- ONE TASK FOR EACH TRANSFORMATION.
- IMPLEMENT STREAMS WITH RENDEZVOUS.
  - TO SEND DATA, CALL AN ENTRY OF THE CONSUMING TASK AND PASS THE DATA ITEM AS AN in PARAMETER.
  - TO RECEIVE DATA, ACCEPT THE ENTRY AND EXAMINE THE PARAMETER.
- ONE RENDEZVOUS FOR EACH ITEM IN THE STREAM. (NO BUFFERING. ONE TASK WAITS UNTIL THE OTHER IS READY.)
- OPTIONAL ENHANCEMENT:
  - CALL A DIFFERENT ENTRY TO SIGNAL THE END OF THE DATA STREAM.
  - RECEIVE DATA WITH A SELECTIVE WAIT:

```
select
   accept Deliver_Item (Item : in Item_Type) do
      Input_Data := Item;
   end Deliver_Item;
or
   accept Report_End_Of_Stream;
   End_Of_Stream := True;
end Select;
```

INSTRUCTOR NOTES

THE MAIN PROGRAM CONSISTS OF A NULL STATEMENT. THE TWO TASKS IMPLEMENTING TRANSFORMATIONS ARE RESPONSIBLE FOR ALL THE COMPUTATION.

THE MAIN PROGRAM TERMINATES ONLY AFTER ITS TWO DEPENDENT TASKS HAVE TERMINATED.

THE SUBUNITS FOR THE TASK BODIES ARE GIVEN ON THE NEXT TWO SLIDES.

Ada SOLUTION TO REFORMATTING PROBLEM

```
procedure Reformat is

   task Line_Disassembler;

   task Line_Assembler is
      entry Deliver_Character (Char : in Character);
   end Line_Assembler;

   task body Line_Disassembler is separate;
   task body Line_Assembler is separate;

begin -- Reformat

   null;  -- ALL WORK DONE BY THE TASKS DECLARED ABOVE

end Reformat;
```

INSTRUCTOR NOTES

THIS IS DERIVED DIRECTLY FROM THE TOP HALF OF SLIDE 15-5, WITH THE SEND OPERATION IMPLEMENTED AS AN ENTRY CALL.

Ada SOLUTION (CONTINUED)

```
with Read_Line;

separate (Reformat)

task body Line_Disassembler is
   Input_Buffer : String (1 .. 80);
begin
   Main_Loop:
      loop
         Read_Line (Input_Buffer);
         for I in Input_Buffer'Range loop
            Line_Assembler.Deliver_Character (Input_Buffer (I));
            exit Main_Loop when Input_Buffer (I) = ASCII.NUL;
         end loop;
      end loop Main_Loop;
end Line_Disassembler;
```

INSTRUCTOR NOTES

THIS IS DERIVED DIRECTLY FROM THE BOTTOM HALF OF SLIDE 15-5, WITH THE RECEIVE OPERATION IMPLEMENTED AS AN ACCEPT STATEMENT.

```
with Write_Line;

separate (Reformat)

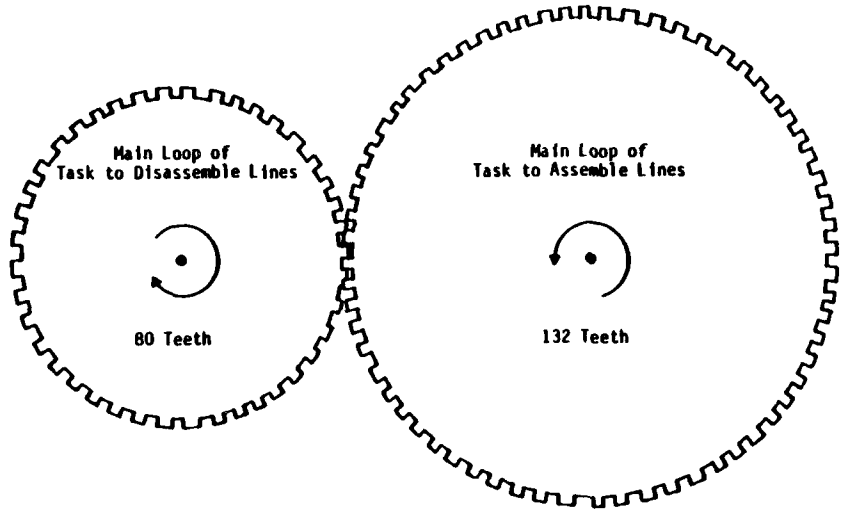
task body Line_Assembler is
   Output_Buffer : String (1 .. 132);
   J             : Integer range Output_Buffer'Range;
begin
   Main_Loop:
      loop
         for I in Output_Buffer'Range loop
            accept Deliver_Character (Char : in Character) do
               Output_Buffer (I) := Char;
            end Deliver_Character;
            if Output_Buffer (I) = ASCII.NUL then
               J := I;
               exit Main_Loop;
            end if;
         end loop;
         Write_Line (Output_Buffer);
      end loop Main_Loop;
   for I in J + 1 .. Output_Buffer'Last loop
      Output_Buffer (I) := ASCII.NUL;
   end loop;
   Write_Line (Output_Buffer);
end Line_Assembler;
```

INSTRUCTOR NOTES

THE OUTER LOOPS OF THE TWO TASKS CYCLE AT DIFFERENT RATES, BUT IN CLOSE COORDINATION. SINCE THERE IS ONE ENTRY CALL IN EACH ITERATION OF Line_Disassembler's INNER LOOP AND ONE ACCEPT STATEMENT IN EACH ITERATION OF Line_Assembler's INNER LOOP, THERE IS ONE ITERATION OF THE FORMER FOR EACH ITERATION OF THE LATTER. HOWEVER, EACH ITERATION OF Line_Disassembler's OUTER LOOP ENTAILS 80 ITERATIONS OF ITS INNER LOOP; EACH ITERATION OF Line_Assembler's OUTER LOOP ENTAILS 132 ITERATIONS OF ITS INNER LOOP. THUS THE GEARS TURN AT DIFFERENT RATES.

STREAM-ORIENTED TASKS ARE TIGHTLY SYNCHRONIZED

- PRODUCER ADVANCES ONE STEP WHEN CONSUMER ADVANCES ONE STEP.



- EACH TOOTH REPRESENTS ONE RENDEZVOUS (ONE ITERATION OF THE INNER LOOP).

- A FULL ROTATION OF THE GEAR REPRESENTS COMPLETE EXECUTION OF A for LOOP (ONE ITERATION OF THE OUTER LOOP).

INSTRUCTOR NOTES

THIS PROBLEM IS TAKEN FROM HOARE'S PAPER ON COMMUNICATING SEQUENTIAL PROCESSES (SEE BIBLIOGRAPHY). HOARE, IN TURN, ATTRIBUTES IT TO THE FOLLOWING PAPER (WHICH INTRODUCED THE TERM COROUTINE):

CONWAY, M.E. DESIGN OF A SEPARABLE TRANSITION-DIAGRAM COMPILER. COMMUNICATIONS OF THE ACM 6, No. 7 (JULY 1963), 396-408.

THE NEXT FEW SLIDES ILLUSTRATE HOW THE ENHANCEMENT AFFECTS THE TWO SOLUTIONS.

AN ENHANCEMENT TO THE REFORMATTER

- BESIDES EXPANDING 80-COLUMN TEXT TO FILL 132 COLUMNS, THE REFORMATTER SHOULD TRANSLATE EACH OCCURRENCE OF "**" IN THE INPUT TO "↑" IN THE OUTPUT.
- THE TRANSLATION SHOULD TAKE PLACE EVEN IF THE "**" IS SPLIT ACROSS LINES.
- THIS ENHANCEMENT MAKES THE TRADITIONAL SOLUTION MUCH MORE COMPLICATED.
- THE STREAM-ORIENTED SOLUTION IS EASILY ADAPTED TO THE NEW REQUIREMENTS.

INSTRUCTOR NOTES

- BULLET 1:

  THIS IS THE HEART OF THE ORIGINAL SOLUTION ON SLIDE 15-4. IT IS GIVEN HERE JUST TO REFRESH STUDENTS' MEMORIES. DO NOT SPEND TIME ON IT.

- BULLET 2:

  THE SOLUTION IS NOW VERY COMPLEX, BECAUSE WE TRY TO DEAL WITH LINE DISASSEMBLY AND TRANSLATION AT THE SAME TIME.

  THE NEXT SLIDE SHOWS THE RESULTING CODE.

EFFECT ON TRADITIONAL SOLUTION

- ORIGINAL FORMULATION

```
Main_Loop:
   loop
      Read_Line (Input_Buffer);
      for I in Input_Buffer'Range loop
         Next_Character := Input_Buffer (I);
         exit Main_Loop when Next_Character = ASCII.NUL;
         Output_Buffer (Output_Position) := Next_Character;
         if Output_Position = Output_Buffer'Last then
            Write_Line (Output_Buffer);
            Output_Position := 1;
         else
            Output_Position := Output_Position + 1;
         end if;
      end loop;
   end loop Main_Loop;
```

- MODIFICATIONS:
  - WHEN A '*' IS FOUND, LOOK AHEAD TO THE NEXT CHARACTER.
    - IF '*': OUTPUT '↑' AND ADVANCE TWO CHARACTERS
    - OTHERWISE: OUTPUT '*' AND ADVANCE ONE CHARACTER
  - NASTY SPECIAL CASES:
    - FIRST '*' IN COLUMN 79: ADVANCING TWO SPACES REQUIRES ANOTHER LINE TO BE READ, ADVANCING ONE SPACE DOES NOT
    - FIRST '*' IN COLUMN 80: LOOKING AHEAD REQUIRES THE NEXT LINE TO BE READ
    - AT THE SAME TIME, KEEP TRACK OF WHETHER LOOK AHEAD IS NECESSARY AT ALL AND WHETHER ADVANCING "NORMALLY" REQUIRES ANOTHER LINE TO BE READ

INSTRUCTOR NOTES

THE MAMMOTH if STATEMENT AT THE TOP OF THE LOOP HANDLES ALL THE CASES DISCUSSED ON THE PREVIOUS SLIDE, SETTING Next_Character AND Input_Position. THE ASSIGNMENT TO Output_Buffer AND BOTTOM if STATEMENT ASSEMBLE OUTPUT LINES.
WARNING!!! THE TOP if STATEMENT IS EXTREMELY COMPLEX, AND IT WOULD BE FOOLHARDY TO TRY TO GO OVER IT IN CLASS. THE ONLY PURPOSE OF THIS SLIDE IS TO CONVINCE STUDENTS THAT THE SOLUTION IS COMPLEX, NOT TO EXPLAIN THE SOLUTION.

HERE IS A SAFETY NET IN CASE YOUR STUDENTS FORCE YOU TO WALK THE TIGHTROPE:

```
if Input_Buffer (Input_Position) = '*' then
   -- Lookahead required
   if Input_Position = Input_Buffer'Last then
      -- A new line must be read to look ahead
      if Input_Buffer (1) = '*' then
         -- Translate ** in columns 80 and 1 into ↑ and advance to column 2
      else
         -- Pass along the * from Column 80 and stay at column 1 for next time
      end if;
   else
      -- Lookahead is on the same line
      if Input_Buffer (Input_Position + 1) = '*' then
         -- Translate ** into ↑ and advance two positions
         if Input_Position = Input_Buffer'Last - 1 then
            -- ** was in columns 79 and 80, so advance to start of next line
         else
            -- Advance two positions in the current line
         end if;
      else
         -- Pass along the * and advance to next position (on same line)
      end if;
else
   -- No lookahead required, just pass along the current character
   if Input_Position = Input_Buffer'Last then
      -- Advance to start to next line
   else
      -- Advance to next position in current line
   end if;
end if;
```

```
Read_Line (Input_Buffer);
Input_Position := 1;
while Input_Buffer (Input_Position) /= ASCII.NUL loop
   if Input_Buffer (Input_Position) = '*' then
      if Input_Position = Input_Buffer'Last then
         Read_Line (Input_Buffer);
         if Input_Buffer (1) = '*' then
            Next_Character := '↑';
            Input_Position := 2;
         else
            Next_Character := '*';
            Input_Position := 1;
         end if;
      else
         if Input_Buffer (Input_Position + 1) = '*' then
            Next_Character := '↑';
            if Input_Position = Input_Buffer'Last - 1 then
               Read_Line (Input_Buffer);
               Input_Position := 1;
            else
               Input_Position := Input_Position + 2;
            end if;
         else
            Next_Character := '*';
            Input_Position := Input_Position + 1;
         end if;
   else
      Next_Character := Input_Buffer (Input_Position);
      if Input_Position = Input_Buffer'Last then
         Read_Line (Input_Buffer);
         Input_Position := 1;
      else
         Input_Position := Input_Position + 1;
      end if;
   end if;
   Output_Buffer (Output_Position) := Next_Character;
   if Output_Position = Output_Buffer'Last then
      Write_Line (Output_Buffer);
      Output_Position := 1;
   else
      Output_Position := Output_Position + 1;
   end if;
end loop;
```

INSTRUCTOR NOTES

- BULLET 2:

  THE MODIFICATION TO THE OVERALL SYSTEM IS SIMPLE.

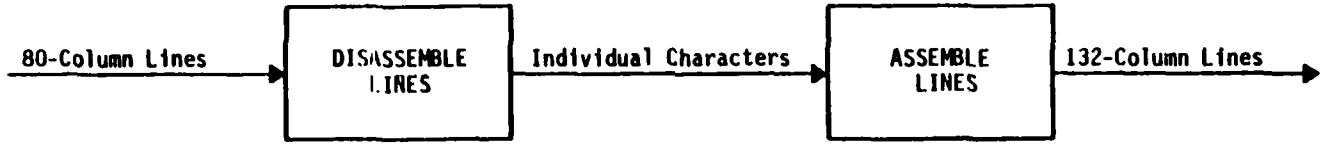
- BULLET 3:

  THE ADDITIONAL TRANSFORMATION IS SIMPLE.

ACTUAL CODE CHANGES ARE GIVEN ON THE NEXT SLIDE.

MODIFICATION TO STREAM_ORIENTED DESIGN

- ORIGINAL DESIGN:



- MODIFIED DESIGN:



- THE "TRANSLATE" TRANSFORMATION SIMPLY COPIES ITS INPUT STREAM TO ITS OUTPUT STREAM, REPLACING EACH "**" WITH "↑".

  - THE TRANSFORMATION IS OBLIVIOUS TO WHERE IN AN INPUT LINE CHARACTERS COME FROM OR WHERE IN AN OUTPUT LINE CHARACTERS GO TO.

INSTRUCTOR NOTES

- BULLET 3:

  THE LOOP REPEATEDLY READS A NON-'*' AND PASSES IT ALONG OR READS A '*' AND THEN A SECOND CHARACTER. IF THE SECOND CHARACTER IS A '*', A '↑' IS PASSED ALONG. OTHERWISE A '*' AND THE SECOND CHARACTER ARE PASSED ALONG.

THE SIMPLICITY OF THIS APPROACH COMES FROM THE FACT THAT IT IS ONLY CONCERNED WITH TRANSLATION OF A STREAM OF CHARACTERS, AND NOT WITH INPUT OR OUTPUT LINES. THE LOGIC OF LOOKING AHEAD AND TRANSLATING IS DECOUPLED FROM THE LOGIC OF ADVANCING TO A NEW INPUT LINE OR A NEW OUTPUT LINE.

MODIFICATION TO Ada TEXT

- ADD TO DECLARATIVE PART OF reformat (THE MAIN PROGRAM):

```
task Translator is
   entry Deliver_Character (Char : in Character);
end Translator;

task body Translator is separate;
```

- IN Line_Disassembler SUBUNIT, REPLACE THE CALL ON Line_Assembler.Deliver_Character WITH A CALL ON Translator.Deliver_Character.

- ADD THE FOLLOWING SUBUNIT:

```
separate (Reformat)

task body Translator is
   Next_Character: Character;
begin
   loop
      accept Deliver_Character (Char : in Character) do
         Next_Character := Char;
      end Deliver_Character;
      if Next_Character = '*' then
         accept Deliver_Character (Char : in Character) do
            Next_Character := Char;
         end Deliver_Character;
         if Next_Character = '*' then
            Line_Assembler.Deliver_Character ('↑');
         else
            Line_Assembler.Deliver_Character ('*');
            Line_Assembler.Deliver_Character (Next_Character);
         end if;
      else
         Line_Assembler.Deliver_Character (Next_Character);
      end if;
      exit when Next_Character = ASCII.NUL;
   end loop;
end Translator;
```

INSTRUCTOR NOTES

- BULLET 2: THIS DOES NOT MEAN THAT PERFORMANCE WILL BE ADVERSELY AFFECTED BY A LARGE NUMBER OF TASKS, ONLY THAT FOR SOME IMPLEMENTATIONS IT MIGHT BE.

- BULLET 3: EMPHASIZE THE "if".

  CALL INVERSION A "MANIPULATION" RATHER THAN A "TRANSFORMATION" SINCE THIS SECTION USES THE TERM "TRANSFORMATION" TO MEAN AN ENTITY THAT MAPS INPUT STREAMS TO OUTPUT STREAMS.

- BULLET 4: EACH IMPLEMENTATION CAN DEFINE ITS OWN PRAGMAS TO SUPPLEMENT THE LANGUAGE-DEFINED PRAGMAS. THE HILFINGER REFERENCE IN THE BIBLIOGRAPHY PROPOSES SUCH PRAGMAS.

  AN UNPUBLISHED PAPER BY NASSI AND HABERMANN DESCRIBES CIRCUMSTANCES IN WHICH A COMPILER CAN MAKE A SIMILAR OPTIMIZATION WITHOUT AID OF A PRAGMA.

- BULLET 5: THE STREAM/TRANSFORMATION MODEL CAN HELP DESIGNERS AND PROGRAMMERS TO UNDERSTAND A COMPLEX PROBLEM. GIVEN A STREAM-ORIENTED SOLUTION, NO CREATIVE THOUGHT IS NECESSARY TO MERGE TASKS -- JUST FOLLOW THE RECIPE.

  EMPHASIZE THE "if" IN THE SECOND ITEM.

STREAM-ORIENTED TASK DESIGN AND EFFICIENCY

- STREAM-ORIENTED TASK DESIGN DECOMPOSES A SEQUENTIAL PROBLEM INTO A POTENTIALLY LARGE NUMBER OF TASKS.
- FOR SOME Ada IMPLEMENTATIONS, THIS MAY SERIOUSLY AFFECT PERFORMANCE.
- IF THIS TURNS OUT TO BE A PROBLEM, THERE ARE MECHANICAL MANIPULATIONS FOR MERGING A SERIES OF TRANSFORMATIONS INTO A SINGLE TASK.
  - THIS MANIPULATION IS ESSENTIALLY JACKSON'S "PROGRAM INVERSION"
  - IT IS DESCRIBED IN SECTION 24
- SOME COMPILERS MAY BE CLEVER ENOUGH TO PERFORM THIS MANIPULATION INTERNALLY.
  - PRAGMAS COULD ALERT THE COMPILER TO THE FACT THAT CERTAIN TASKS CAN BE MERGED.
  - THESE TASKS WILL BE COMPILED AS COROUTINES, WITHOUT EXPENSIVE "CONTEXT SWITCHES" WHEN ONE TASK PASSES CONTROL TO ANOTHER.
  - ONLY THE OBJECT CODE WOULD BE AFFECTED. THE SIMPLICITY AND CLARITY OF THE SOURCE CODE WOULD BE PRESERVED.
- WHETHER OR NOT TASKS MUST BE MERGED TO MEET PERFORMANCE REQUIREMENTS, STREAM-ORIENTED TASK DESIGN IS USEFUL AS A DESIGN TOOL.
  - FIRST WRITE A CLEAR, SIMPLE, BUT POTENTIALLY INEFFICIENT SOLUTION.
  - IF NECESSARY, MECHANICALLY MANIPULATE THIS SOLUTION TO OBTAIN A LESS CLEAR BUT MORE EFFICIENT SOLUTION.

INSTRUCTOR NOTES

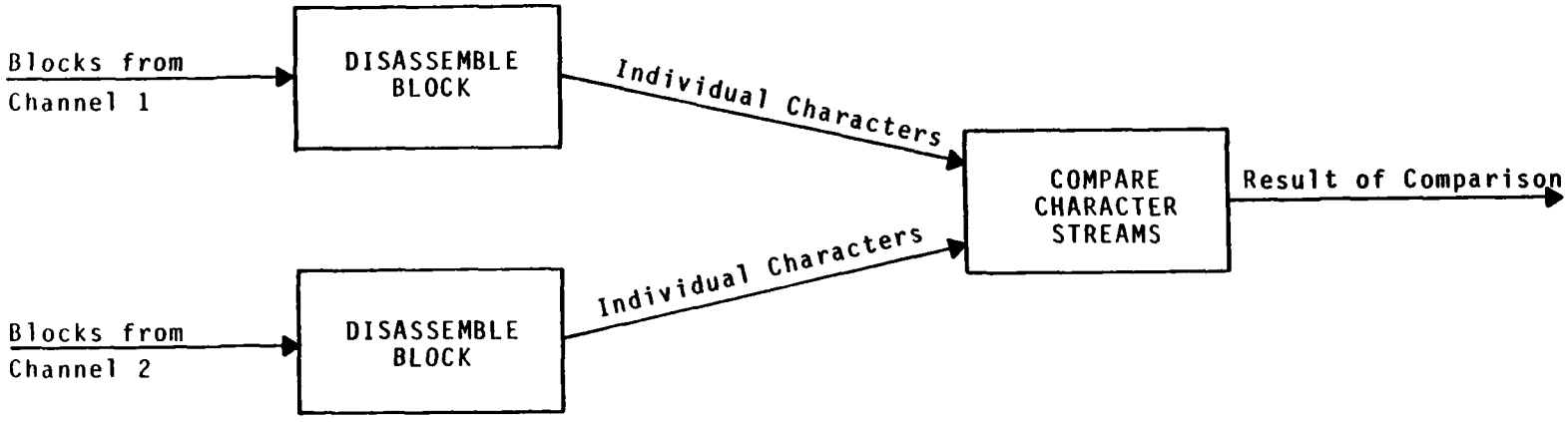
THIS IS THE PROMISED SOLUTION TO THE PROBLEM GIVEN ON SLIDE 15-1.

THE COMPARISON WORKS WITH STREAMS CHARACTERS, OBLIVIOUS TO THE WAY THOSE CHARACTERS WERE PARTITIONED INTO BLOCKS.

THE "OUTPUT STREAM" OF THE COMPARISON CONSISTS OF A SINGLE Boolean VALUE.

THE NEXT TWO SLIDES GIVE THE Ada TEXT OF THE SOLUTION.

STREAM-ORIENTED SOLUTION TO THE MESSAGE PROBLEM



THE COMPARISON TRANSFORMATION IS PERFORMED BY THE FUNCTION'S SEQUENCE OF STATEMENTS. THE TWO TASKS Channel_1_Disassembler AND Channel_2_Disassembler PERFORM THE BLOCK DISASSEMBLY TRANSFORMATIONS. THE SUBUNIT FOR THE Block_Disassembler_Type TASK BODY IS GIVEN ON THE NEXT SLIDE.

THE DIRECTION OF THE RENDEZVOUS IS THE OPPOSITE OF WHAT WE HAVE SEEN BEFORE. THE COMPARISON TRANSFORMATION CALLS THE Get_Character ENTRY OF A BLOCK DISASSEMBLY TASK TO RECEIVE A CHARACTER. A BLOCK DISASSEMBLY TASK SENDS THAT CHARACTER BY ASSIGNING TO THE FORMAL PARAMETER IN AN ACCEPT STATEMENT.

THIS ALLOWS THE COMPARISON TRANSFORMATION TO CONTROL WHICH STREAM IT RECEIVES A CHARACTER FROM. (SEE THE FIRST TWO STATEMENTS IN THE Same_Message LOOP.) IF THE Block_Disassembler_Type TASK BODY SENT A MESSAGE BY CALLING A PARTICULAR ENTRY OF A COMPARISON TASK, THE COMPARISON TASK WOULD NOT BE ABLE TO CONTROL WHICH STREAM IT OBTAINED A CHARACTER FROM.

THE Assign_Channel ENTRIES ARE USED TO INITIALIZE THE BLOCK DISASSEMBLY TASKS, BY TELLING EACH WHICH CHANNEL IT SHOULD READ FROM.

THE FIRST if STATEMENT IN THE LOOP ACCOMMODATES THE CASE WHERE ONE OF THE CHARACTERS BEING COMPARED IS A NULL BYTE AND THE OTHER IS NOT. THIS HAPPENS WHEN ONE MESSAGE IS LARGER THAN THE OTHER, BUT THE MESSAGES MATCHED THROUGHOUT THE LENGTH OF THE SHORTER MESSAGE.

```
with Block_Package;

function Same_Message return Boolean is

   Channel_1_Char, Channel_2_Char : Character;

   task type Block_Disassembler_Type is
      entry Assign_Channel (Channel : in Block_Package.Channel_Number_Type);
      entry Get_Character (Char : out Character);
   end Block_Disassembler_Type;
   Channel_1_Disassembler, Channel_2_Disassembler : Block_Disassembler_Type;
   task body Block_Disassembler_Type is separate;
begin -- Same_Message
   Channel_1_Disassembler.Assign_Channel (1);
   Channel_2_Disassembler.Assign_Channel (2);
   loop
      Channel_1_Disassembler.Get_Character (Channel_1_Char);
      Channel_2_Disassembler.Get_Character (Channel_2_Char);
      if Channel_1_Char /= Channel_2_Char then
         return False;
      end if;
      if (Channel_1_Char = ASCII.NUL and Channel_2_Char = ASCII.NUL) then
         return True;
      end if;
   end loop;
end Same_Message;
```

INSTRUCTOR NOTES

Block_Package WAS DECLARED ON SLIDE 15-1.

THE terminate ALTERNATIVE IS NEEDED TO ALLOW THE Same_Message FUNCTION TO RETURN WHEN A MISMATCH IS FOUND. (THE FUNCTION IS THE MASTER OF THE TWO TASKS.) IT WOULD NOT DO FOR THE DISASSEMBLY TASK TO TEST FOR A NULL BYTE AND THEN TERMINATE, BECAUSE THE FUNCTION RETURNS WITHOUT READING BOTH STREAMS IN THEIR ENTIRETY WHEN A MISMATCH IS FOUND.

Block_Disassembler_Type SUBUNIT

```
separate (Same_Message)

task body Block_Disassembler_Type is

   My_Channel   : Block_Package.Channel_Number_Type;
   BLock_Buffer : Block_Package.Block_Buffer_Subtype;
   Number_Read  : Block_Package.Character_Count_Subtype;

begin

   accept Assign_Channel (Channel : in Block_Package.Channel_Number_Type) do
      My_Channel := Channel;
   end Assign_Channel;

   loop
      Block_Package.Get_Next_Block
         (From_Channel => My_Channel,
          Into_Buffer  => Block_Buffer,
          Block_Size   => Number_Read);
      for I in 1 .. Number_Read loop
         select
            accept Get_Character (Char : out Character) do
               Char := Block_Buffer (I);
            end Get_Character;
         or
            terminate;
         end select;
      end loop;  -- for I
   end loop;     -- main loop
end Block_Disassembler_Type;
```

INSTRUCTOR NOTES

BEGIN THIS EXERCISE AT THE START OF DAY 4 AND ALLOW 90 MINUTES, INCLUDING TIME TO GO OVER THE SOLUTION.

IT MIGHT BE WISE TO IMPOSE A TIME LIMIT FOR EACH PART OF THE PROBLEM AND TO PRESENT SOLUTIONS TO EACH PART BEFORE LETTING STUDENTS GO ON TO THE NEXT PART. THIS WILL PREVENT STUDENTS FROM GETTING OFF ON THE WRONG TRACK AND WILL GIVE THOSE WHO GET STUCK ON ONE PART A CHANCE TO TRY LATER PARTS.

FOR PART 3 TELL STUDENTS TO ASSUME THE EXISTENCE OF THE FOLLOWING PROCEDURES:

```
Procedure Display_Position (Position : in Position_Type);
Procedure Downlink_Position (Position : in Position_Type);
```

"DOWNLINKING" MEANS TRANSMITTING DATA FROM THE AIRCRAFT TO THE GROUND.

EXERCISE 15.1

AN AIRCRAFT NAVIGATION PROGRAM COMPUTES CURRENT POSITIONS FROM SENSOR READINGS, DISPLAYS CURRENT POSITIONS TO THE PILOT, AND "DOWNLINKS" CURRENT POSITIONS TO THE GROUND.

THE PROGRAM OBTAINS A PACKET OF THREE READINGS BY CALLING THE ENTRY Multiplexer_Task.Get_Packet WITH A Packet_Type PARAMETER. Packet_Type IS AN ARRAY TYPE WITH THREE Float COMPONENTS.

THE PROGRAM REPEATEDLY USES THE FIVE MOST RECENT SENSOR READINGS (WHICH MAY HAVE COME FROM TWO OR THREE DIFFERENT PACKETS) TO COMPUTE AN AVERAGED SENSOR READING.

THE FOLLOWING FUNCTION TAKES AN AVERAGED SENSOR READING AND COMPUTES THE CURRENT POSITION OF THE AIRCRAFT:

```
function Updated_Position
   (Previous_Position: Position_Type; Averaged_Reading: Float)
    return Position_Type;
```

THE PARAMETERLESS FUNCTION Initial_Position RETURNS THE POSITION OF THE AIRCRAFT WHEN THE NAVIGATION SYSTEM IS FIRST ACTIVATED. POSITIONS ARE COMPUTED FROM EACH AVERAGED READING. AFTER EVERY ELEVEN POSITION COMPUTATIONS, THE PROCEDURE

```
Correct_Position (Position : in out Position_Type);
```

MUST BE CALLED (BETWEEN CALLS ON Updated_Position AND BEFORE THE POSITION IS DISPLAYED) TO CORRECT FOR THE CURVATURE OF THE EARTH.

EVERY POSITION THAT HAS BEEN COMPUTED BY Update_Position AND, IF NECESSARY, CORRECTED BY Correct_Position IS DISPLAYED ONBOARD THE AIRCRAFT. EVERY SEVENTH POSITION IS DOWNLINKED.

THIS EXERCISE HAS THREE PARTS:

1. DRAW A STREAM/TRANSFORMATION DIAGRAM OF A SOLUTION TO THIS PROBLEM. (GIVEN AN INPUT STREAM OF PACKETS, WHAT TRANSFORMATIONS ARE NECESSARY TO PRODUCE AN OUTPUT STREAM OF POSITIONS TO BE DISPLAYED AND AN OUTPUT STREAM OF POSITIONS TO BE DOWNLINKED?)
2. WRITE THE TASK DECLARATIONS FOR EACH TRANSFORMATION IN YOUR DIAGRAM.
3. WRITE THE BODIES FOR AS MANY OF THESE TASKS AS YOU HAVE TIME FOR.

INSTRUCTOR NOTES

PART V

OTHER TASKING FEATURES

16. ABORTING TASKS

17. EXCEPTIONS IN MULTITASK PROGRAMS

18. INTERRUPT ENTRIES

19. ENTRY FAMILIES

20. TASK PRIORITIES

INSTRUCTOR NOTES

ALLOW 30 MINUTES FOR THIS SECTION.

RECOMMEND THAT STUDENTS READ EXERCISE 5.1 IN THE REAL-TIME ADA WORKBOOK.

Section 16

ABORTING TASKS

INSTRUCTOR NOTES

- BULLET 2:

  BE CAREFUL NOT TO SAY THAT AN abort STATEMENT "ABORTS" A TASK OR CAUSES "UNCONDITIONAL" TERMINATION. THE NEXT SLIDE CONTRADICTS THIS.

- BULLET 3:

  A TASK NAME MAY BE THE IDENTIFIER OF A DECLARED TASK OBJECT, OR IT MAY NAME A COMPONENT OF AN ARRAY OF TASKS, A COMPONENT OF A RECORD, OR AN ALLOCATED OBJECT DESIGNATED BY AN ACCESS VALUE.
  A TASK MAY NAME ITSELF IN AN abort STATEMENT.

- BULLET 4:

  SLIDE 16-5 WILL DISCUSS WHY THE abort STATEMENT IS UNSUITABLE FOR ROUTINE USE.

THE abort STATEMENT

- THE abort STATEMENT IS AN "EMERGENCY BRAKE".
- IT IS USED TO PREVENT AN OUT-OF-CONTROL TASK FROM INTERFERING WITH THE REST OF A PROGRAM.
- FORM:

abort [task name] {, [task name] };

- THIS STATEMENT IS INTENDED FOR USE ONLY IN EXTREME SITUATIONS, NOT FOR ROUTINE TASK MANAGEMENT.

INSTRUCTOR NOTES

- BULLET 2:

  - ITEM 1: THE TERM COMPLETED IS A TECHNICAL TERM WITH A PRECISE DEFINITION. THE TERM "INTERACT" IS USED INFORMALLY TO REFER TO ANY OF THE ACTIONS THAT WILL BE LISTED ON SLIDE 16-4.

  - ITEM 3:

    - SUBITEM 3: IN A NETWORK, FOR EXAMPLE, "ABORTING" A TASK MIGHT CONSIST OF DISCONNECTING THE PROCESSOR RUNNING THE TASK FROM THE NETWORK. IT WOULD BE IRRELEVANT TO THE REST OF THE NETWORK WHETHER THE TASK EVER COMPLETED.

SEMANTICS OF THE abort STATEMENT

- TECHNICALLY, AN abort STATEMENT DOES NOT ABORT A TASK!
  - IT CAUSES THE TASK TO BECOME ABNORMAL.
- AN ABNORMAL TASK IS "EXCOMMUNICATED"
  - AS SOON AS IT TRIES TO INTERACT WITH OTHER TASKS, AN ABNORMAL TASK BECOMES COMPLETED.
    - *"INTERACTING" INCLUDES MORE THAN JUST RENDEZVOUS (SEE NEXT SLIDE).*
    - COMPLETED MEANS DONE EXECUTING AND READY TO TERMINATE AS SOON AS DEPENDENT TASKS ARE TERMINATED.
  - IF AN ABNORMAL TASK IS STUCK IN A LOOP WHERE IT NEVER INTERACTS WITH ANOTHER TASK AND NEVER RAISES AN EXCEPTION, IT NEED NOT EVER BECOME COMPLETED.
    - DEPENDS ON THE IMPLEMENTATION
    - A RUNTIME SYSTEM WITH PREEMPTIVE SCHEDULING WILL PROBABLY COMPLETE THE TASK
    - A RUNTIME SYSTEM FOR DISTRIBUTED PROCESSORS MAY NOT

INSTRUCTOR NOTES

- BULLET 2:

SINCE THE DEPENDENT TASKS WILL EVENTUALLY COMPLETE, THOSE AT THE BOTTOM LEVEL OF THE "DEPENDENCY TREE" WILL TERMINATE. THEN THE TASKS AT THE NEXT HIGHER LEVEL WILL BE ABLE TO TERMINATE, AND SO ON.

FOR TYPICAL IMPLEMENTATIONS, THE abort STATEMENT DOES EVENTUALLY ABORT A TASK AND ITS DEPENDENTS; BUT THE RULES OF Ada DO NOT REQUIRE THIS.

ABORTING A MASTER

- WHEN A TASK BECOMES ABNORMAL, SO DO ALL ITS DEPENDENT TASKS.

- IF, FOR A GIVEN IMPLEMENTATION, ABNORMAL TASKS ALWAYS BECOME COMPLETE, THEN THE abort STATEMENT WILL ALWAYS CAUSE THE NAMED TASKS (AND ITS DEPENDENT TASKS) TO TERMINATE (EVENTUALLY).

INSTRUCTOR NOTES

THIS SLIDE LISTS EVENTS THAT FORCE AN ABNORMAL TASK TO BECOME COMPLETED. EXCEPT AS NOTED IN BULLET 3, A RUNTIME SYSTEM MAY CAUSE AN ABNORMAL TASK TO BECOME COMPLETED AT ANY OTHER TIME AS WELL.

AN ABNORMAL TASK CAN BE STUCK IN A LOOP IN WHICH NONE OF THESE EVENTS OCCUR.

- BULLET 1:

  LIKE ANY OTHER TASK, AN ABNORMAL TASK ALSO BECOMES COMPLETED UPON REACHING THE END OF ITS SEQUENCE OF STATEMENTS.

  - ITEM 3: IF A TASK BECOMES ABNORMAL WHILE WAITING FOR ITS ENTRY CALL TO BE ACCEPTED, THE ENTRY CALL IS CANCELLED (REMOVED FROM THE QUEUE).

  - ITEM 7:

    SUBITEM 2: REACHING THE begin OF A TASK BODY (WHETHER OR NOT IT FOLLOWS A TASK OBJECT DECLARATION) IS COVERED BY ITEM 2 (COMPLETING ELABORATION OF A TASK BODY DECLARATIVE PART).

    THIS SUBITEM ALSO APPLIES TO THE begin IN A SUBPROGRAM OR BLOCK STATEMENT EXECUTED, OR A PACKAGE BODY ELABORATED, BY THE TASK.

- BULLET 3: THE LANGUAGE RULES REFLECT THE IDEA OF SERVER AND USER TASKS PRESENTED IN SECTION 11.

- BULLET 4: THIS CORRESPONDS TO ITEMS 1, 3, 4, AND 5 OF BULLET 1. IN ALL OTHER CASES, A TASK EXECUTING AN abort STATEMENT MAY GO ON EVEN THOUGH THE ABNORMAL TASK IS NOT YET COMPLETE.

WHEN AN ABNORMAL TASK MUST BECOME COMPLETED

- UPON CERTAIN EVENTS AFFECTING TASK ACTIVATION AND INTERACTION:
  - BEGINNING ELABORATION OF TASK BODY DECLARATIVE PART (OR CONTINUING TO WAIT FOR IT TO BEGIN).
  - COMPLETING ELABORATION OF TASK BODY DECLARATIVE PART.
  - REACHING (OR CONTINUING TO WAIT AT) AN ENTRY CALL.
  - REACHING (OR CONTINUING TO WAIT AT) A delay STATEMENT.
  - REACHING (OR CONTINUING TO WAIT AT) A SELECTIVE WAIT OR AN accept STATEMENT.
  - REACHING THE END OF AN accept STATEMENT.
  - CAUSING ANOTHER TASK TO BECOME ACTIVE.
    - ALLOCATING A TASK OBJECT
    - REACHING THE begin FOLLOWING A TASK OBJECT DECLARATION
  - REACHING ANOTHER abort STATEMENT.
- UPON ENTERING AN EXCEPTION HANDLER
- A CALLING TASK THAT BECOMES ABNORMAL WHILE IN A RENDEZVOUS MUST NOT BECOME COMPLETED UNTIL AFTER THE RENDEZVOUS IS OVER.
  - THE accept STATEMENT COMPLETES NORMALLY.
  - THE CALLED TASK REMAINS ALIVE AND WELL TO SERVE OTHER CALLERS.
- A TASK THAT BECOMES ABNORMAL WHILE WAITING FOR ACTIVATION, RENDEZVOUS, OR EXPIRATION OF A DELAY BECOMES COMPLETED IMMEDIATELY.
  - MAY BE ASSUMED TO BE COMPLETED IMMEDIATELY AFTER THE abort STATEMENT THAT MADE THE TASK ABNORMAL.

INSTRUCTOR NOTES

THE NEXT SLIDE SUGGESTS AN ALTERNATIVE TO THE abort STATEMENT.

- BULLET 1:

  AN EXCEPTION IS RAISED IN A TASK THAT TRIES TO CALL AN ENTRY OF AN ABNORMAL TASK. SECTION 17 GIVES DETAILS.

- BULLET 2:

  IF A SYSTEM HAS REDUNDANT SENSORS AND A HARDWARE FAILURE CAUSES THE TASK ASSIGNED TO ONE OF THEM TO GO AWAY, IT MAY BE POSSIBLE TO ABORT THAT TASK AND CONTINUE PROCESSING. MORE OFTEN, HOWEVER, THERE IS NO SENSIBLE WAY TO CONTINUE A PROGRAM ONCE ONE OF ITS TASKS HAS BEEN ABORTED.

- BULLET 3:

  TERMINATION MIGHT NEVER OCCUR, OR IT MIGHT NOT OCCUR IMMEDIATELY. THE SUBBULLETS SHOW THAT EVEN IF ABNORMAL TASKS DO NOT "INTERACT" BY PERFORMING ONE OF THE ACTIONS LISTED ON THE PREVIOUS SLIDE, THEIR CONTINUED EXECUTION CAN STILL HAVE TANGIBLE EFFECTS.

- BULLET 4:

  AN AUDIT TRAIL IS A LOG OF EVENTS AND ACTIONS MAINTAINED BY THE EXECUTING PROGRAM. IT CAN BE USED TO DIAGNOSE FAILURES, TO RECONSTRUCT DATA BASES AFTER A FAILURE, AND TO DETECT BREACHES OF SECURITY, FOR EXAMPLE. IT WOULD BE USEFUL FOR AN ABNORMAL TASK TO RECORD ITS DEMISE IN AN AUDIT TRAIL, BUT THE abort STATEMENT DOES NOT PERMIT THIS.

- BULLET 6:

  - ITEM 2: THIS IS BECAUSE THE LANGUAGE RULES LEAVE MANY OPTIONS OPEN TO THE IMPLEMENTATION.

PROBLEMS WITH THE abort STATEMENT

- IT MAY FOUL UP OTHER TASKS TRYING TO COMMUNICATE WITH THE ABORTED TASK.
  - THIS CAN LEAD TO A RIPPLE EFFECT.
- MOST TASKS ARE INDISPENSABLE TO THE PROGRAM.
- EXECUTING AN abort STATEMENT MIGHT NOT TERMINATE THE TASK.
  - IT MAY CONTINUE TO USE PROCESSOR TIME OR PERFORM I/O
  - IT MAY CONTINUE TO CHANGE GLOBAL VARIABLES
- AN ABORTED TASK HAS NO CHANCE TO FULFILL ITS "LAST WISHES".
  - COMPLETING A DATA STRUCTURE MANIPULATION
    - IF ABORTED IN THE MIDDLE OF REMOVING A CELL FROM A DOUBLY-LINKED LIST, FOR EXAMPLE, IT MAY LEAVE THE CELL LINKED IN ONLY ONE DIRECTION.
  - PERFORMING CLEANUP
    - DEALLOCATING ALLOCATED VARIABLES
    - WRITING AN AUDIT TRAIL
    - CLOSING FILES
- MAKES PROGRAMS HARD TO UNDERSTAND
  - MORE OPPORTUNITIES FOR SUBTLE AND COMPLEX TASK INTERACTION.
- MAKES PROGRAMS HARDER TO VALIDATE
  - FEWER ASSUMPTIONS CAN BE MADE
  - WIDE RANGE OF POSSIBLE STATES AFTER AN abort STATEMENT

INSTRUCTOR NOTES

IN ADDITION TO THE ENTRIES REQUIRED TO DO ITS NORMAL PROCESSING, THIS TASK HAS AN EXTRA ENTRY, Request_Termination, THAT CAN BE USED BY OTHER TASKS TO ASK Sensor_Task TO TERMINATE.

WHEN IT IS IN A SELECTIVE WAIT (AND THUS NOT IN THE MIDDLE OF SOME INDIVISIBLE OPERATION), Sensor_Task MAY ACCEPT A CALL ON Request_Termination. IT RESPONDS TO THIS REQUEST BY PERFORMING ANY NECESSARY CLEAN-UP ACTIONS (INDICATED HERE BY A CALL ON SOME PROCEDURE Clean_Up) AND COMPLETING. IN THIS CASE, COMPLETION IS ACHIEVED BY EXITING FROM THE BASIC LOOP AND COMING TO THE END OF THE TASK BODY'S STATEMENT SEQUENCE.

A MORE CONTROLLED WAY TO ACHIEVE TERMINATION

```
task type Sensor_Task is
   entry ...;
      ...
   entry ...;
   entry Request_Termination;
end Sensor_Task;

task body Sensor_Task is
   ...
begin
   ...
   loop
      select
         accept ...;
      or
      ...
      or
         accept ...;
         ...
      or
         accept Request_Termination;
         Clean_Up;
         exit;
      or
      ...
      end select;

   end loop;

end Sensor_Task;
```

INSTRUCTOR NOTES

- BULLET 4:

  IN A GENERAL-PURPOSE OPERATING SYSTEM CERTAIN TASKS ARE RUNNING PROGRAMS WRITTEN BY POTENTIALLY HOSTILE OR MISCHIEVOUS USERS, SO NOT ALL TASKS CAN BE TRUSTED.

  IN AN EMBEDDED SYSTEM, TASKS ARE PART OF A COHERENT DESIGN AND ARE WRITTEN TO WORK COOPERATIVELY WITH EACH OTHER.

  THE NEXT SLIDE SHOWS WHAT CAN BE DONE IF, FOR SOME REASON (e.g. THE POSSIBILITY OF PROGRAMMING ERROR) THE TASK TO BE TERMINATED CANNOT BE TRUSTED.

"PLEASE TERMINATE" ENTRY

- MORE FLEXIBLE THAN terminate ALTERNATIVES.
  - TASK CAN BE ASKED TO TERMINATE BEFORE ITS MASTER IS COMPLETED.
- GIVES THE TASK TO BE TERMINATED SOME CONTROL OVER ITS DEMISE.
  - TERMINATION CAN ONLY OCCUR WHEN THE TASK IS READY TO TERMINATE (NOT IN THE MIDDLE OF A LINKED LIST UPDATE).
  - THE TASK HAS A CHANCE TO EXECUTE "LAST WISHES".
- MAKES THE SET OF EVENTS UPON PROGRAM TERMINATION AN EXPLICIT PART OF THE PROGRAM TEXT.
- THE TASK TO BE TERMINATED IS TRUSTED.
  - TO ACCEPT A "PLEASE TERMINATE" ENTRY CALL IN A TIMELY MANNER.
  - TO ACT UPON THE REQUEST IN A TIMELY MANNER.

INSTRUCTOR NOTES

- BULLET 1:

  IT IS ASSUMED THAT Sensor_Task_List IS AN ARRAY OF TASKS HANDLING SENSORS AND THAT SOME SELF-TEST HAS IDENTIFIED A MALFUNCTIONING SENSOR AND PLACED ITS INDEX IN Bad_Sensor.

  THE DELAY IS DESIGNED TO GIVE Sensor_Task_List (Bad_Sensor) A CHANCE TO ACCEPT THE ENTRY CALL, PERFORM ANY CLEAN-UP, AND COMPLETE OF ITS OWN ACCORD.

- BULLET 2:

  IT IS OKAY TO EXECUTE AN abort STATEMENT FOR A TASK THAT IS ALREADY ABNORMAL, COMPLETED, OR TERMINATED. IT SIMPLY HAS NO EFFECT.

- BULLET 3:

  THE LANGUAGE DESIGNERS MADE THE INTENDED USE OF THE abort STATEMENT CLEAR IN THE PUBLISHED RATIONALE FOR THE LANGUAGE DESIGN AND IN THE NOTES OF THE REFERENCE MANUAL. THE abort STATEMENT WAS INCLUDED IN THE LANGUAGE ONLY RELUCTANTLY.

THE abort STATEMENT AS A BACKUP MEASURE

- A "PLEASE TERMINATE" ENTRY CAN BE USED AS THE PRIMARY MEANS OF STOPPING A TASK, WITH AN abort STATEMENT USED AS A BACKUP MEASURE:

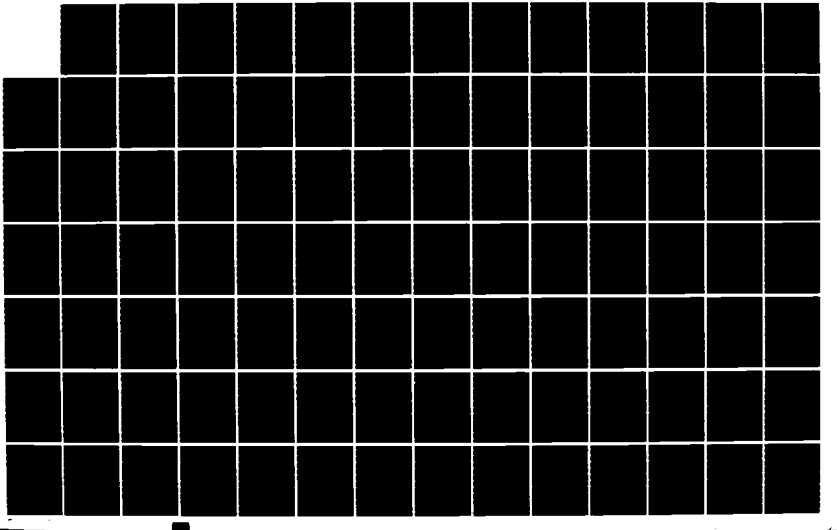
```
Sensor_Task_List (Bad_Sensor).Request_Termination;
delay 30.0; -- Give the task a chance to respond
abort Sensor_Task_List (Bad_Sensor); -- No effect if
                                     -- already terminated
```

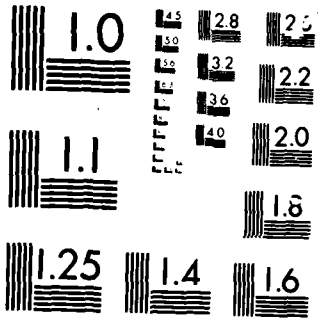
- IF THE CALLED TASK TERMINATES IN A TIMELY MANNER, THE abort STATEMENT HAS NO EFFECT.

- THIS IS THE INTENDED USE OF abort STATEMENTS, AS A MEASURE OF LAST RESORT.

AD-A166 352 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS IN ADA L401 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 3/6

UNCLASSIFIED F/G 5/9 NL





MICROCOPY RESOLUTION TEST CHART

INSTRUCTOR NOTES

- BULLET 3:

A DESIGNER IS DECEIVING HIMSELF IF HE ASSUMES THAT THINGS WILL "TAKE CARE OF THEMSELVES" WHEN A TASK IS ABORTED. RECOVERING FROM TASK FAILURE IS A DIFFICULT PROBLEM THAT MUST BE CONFRONTED EXPLICITLY.

A PROGRAMMER SHOULD NEVER TAKE IT UPON HIMSELF TO ABORT A TASK OR CALL A "PLEASE TERMINATE" ENTRY IF THE DESIGN DOES NOT ANTICIPATE THAT HE WILL DO SO.

SUMMARY

DESIGN HINT

- AVOID THE abort STATEMENT.
- USE NORMAL TASK COMMUNICATION TO GET A TASK TO TERMINATE.
- MAKE THE PROVISION FOR STOPPING TASKS AN INTEGRAL PART OF THE DESIGN.

INSTRUCTOR NOTES

ALLOW 30 MINUTES FOR THIS SECTION.

Section 17

EXCEPTIONS IN MULTITASK PROGRAMS

INSTRUCTOR NOTES

THIS SECTION COVERS BOTH EXCEPTIONS RAISED BY TASKING OPERATIONS AND THE WAY THAT ORDINARY EXCEPTIONS (LIKE Constraint_Error) AFFECT MULTITASK PROGRAMS.

- BULLET 1:
  - ITEM 2: EXCEPTIONS INSIDE accept STATEMENTS ARE COVERED BY THE NEXT ITEM.

OVERVIEW OF EXCEPTIONS IN MULTITASK PROGRAMS

- CONTEXTS IN WHICH EXCEPTIONS MAY BE RAISED:
  - IN THE DECLARATIVE PART OF A TASK BODY
  - IN THE SEQUENCE OF STATEMENTS OF A TASK BODY (OUTSIDE OF AN ACCEPT STATEMENT)
  - DURING INTERTASK COMMUNICATION
    - EXCEPTIONS ARISING INSIDE AN accept STATEMENT
    - ATTEMPT TO CALL A COMPLETED OR ABNORMAL TASK
    - TASK ABORTED DURING A RENDEZVOUS
- DIFFERENT RULES APPLY IN EACH CASE.
- THE PREDEFINED EXCEPTION Tasking_Error IS RAISED IN CERTAIN SITUATIONS INVOLVING TASK INTERACTION.

INSTRUCTOR NOTES

TO UNDERSTAND WHAT HAPPENS WHEN AN EXCEPTION IS RAISED IN THE DECLARATIVE PART OF THE TASK BODY, WE MUST TAKE A CLOSER LOOK AT THE STEPS INVOLVED IN ACTIVATION OF A TASK. IN PARTICULAR, NO SEQUENCE OF STATEMENTS MAY BEGIN UNTIL EACH DECLARATIVE PART HAS BEEN PROCESSED.

- BULLET 1: THE NEXT SLIDE HAS AN EXAMPLE.

  - ITEM 1: THE EXCEPTION RAISED IN STEP 2, IF IT IS NOT TO BE PROPAGATED, CAN ONLY BE HANDLED BY A HANDLER ON THE CORRESPONDING SUBPROGRAM BODY, BLOCK STATEMENT, TASK BODY OR PACKAGE BODY. IT IS IMPOSSIBLE TO ADD A BLOCK STATEMENT SURROUNDING THE POINT AT WHICH THE EXCEPTION IS RAISED, SO THAT THE EXCEPTION CAN BE HANDLED MORE LOCALLY. (HOWEVER, THE TASK OBJECT DECLARATION ITSELF CAN BE MOVED INTO A BLOCK STATEMENT.)

    IN THE CASE OF A PACKAGE, THE SEQUENCE OF STATEMENTS IN STEP 3 IS THE SEQUENCE OF INITIALIZATION STATEMENTS, IF ANY.

    THE FOLLOWING INFORMATION IS FOR THE INSTRUCTOR'S BACKGROUND ONLY. DO NOT PRESENT IT TO THE CLASS.

    IF A TASK IS DECLARED IN A PACKAGE SPECIFICATION WITHOUT A CORRESPONDING PACKAGE BODY, TWO CASES ARISE:

    - IF THE PACKAGE SPECIFICATION IS IN A DECLARATIVE PART, THE TASK BEGINS EXECUTING ITS STATEMENTS AT THE END OF THE ELABORATION OF THAT DECLARATIVE PART. THE EXCEPTION IS RAISED IN THE DECLARATIVE PART.

    - IF THE PACKAGE SPECIFICATION IS A LIBRARY UNIT, THE TASK BEGINS EXECUTING ITS STATEMENT BEFORE THE MAIN PROGRAM COMMENCES. THF EXCEPTION CAUSES THE MAIN PROGRAM TO BE ABANDONED BEFORE IT STARTS.

EXCEPTIONS IN TASK BODY DECLARATIVE PARTS

- WHEN SEVERAL TASK OBJECTS ARE DECLARED IN THE SAME SUBPROGRAM, BLOCK STATEMENT, TASK BODY, OR PACKAGE:
  - THE FOLLOWING EVENTS OCCUR UPON ENTRY TO THE SEQUENCE OF STATEMENTS IN THE SUBPROGRAM, BLOCK STATEMENT, TASK BODY, OR PACKAGE:
    1. EACH OF THE DECLARED TASKS BEGINS ELABORATING THE DECLARATIVE PART OF ITS TASK BODY.
    2. IF ONE OR MORE OF THESE TASKS RAISES AN EXCEPTION WHILE ELABORATING THE DECLARATIVE PART OF ITS TASK BODY, THAT TASK BECOMES COMPLETED, AND Tasking_Error IS RAISED ONCE, AT THE BEGINNING OF THE SEQUENCE OF STATEMENTS IN THE SUBPROGRAM BODY, BLOCK STATEMENT, TASK BODY, OR PACKAGE BODY.
    3. ONCE EACH OF THE DECLARED TASKS HAS ELABORATED THE DECLARATIVE PART OF ITS TASK BODY -- OR DIED TRYING -- THE SURVIVING TASKS AND THE SUBPROGRAM, BLOCK STATEMENT, TASK, OR PACKAGE DECLARING THEM BEGIN CONCURRENT, ASYNCHRONOUS EXECUTION OF THEIR SEQUENCES OF STATEMENTS.
  - THIS APPLIES TO INDIVIDUALLY DECLARED TASK OBJECTS AND TO TASK TYPE COMPONENTS OF DECLARED COMPOSITE OBJECTS.
- WHEN A TASK OBJECT, OR A COMPOSITE OBJECT CONTAINING ONE OR MORE TASK OBJECTS, IS ALLOCATED, THE FOLLOWING EVENTS OCCUR:
  - EACH TASK CREATED BY THE ALLOCATION BEGINS ELABORATING THE DECLARATIVE PART OF ITS TASK BODY.
  - IF ONE OR MORE OF THE ALLOCATED TASKS RAISES AN EXCEPTION WHILE ELABORATING THE DECLARATIVE PART OF ITS TASK BODY, THAT TASK BECOMES COMPLETED. Tasking_Error IS RAISED ONCE, BY THE ALLOCATOR.
  - SURVIVING TASKS BEGIN CONCURRENT, ASYNCHRONOUS EXECUTION OF THEIR SEQUENCES OF STATEMENTS.
  - EVALUATION OF THE ALLOCATOR FINISHES, YIELDING AN ACCESS VALUE.

THIS PROCEDURE DOES NOTHING USEFUL. IT PURPOSELY RAISES EXCEPTIONS TO ILLUSTRATE THE RULES.

THE COMMENT IN THE Hopeless_Task_Type EXCEPTION HANDLER REFERS TO THE FACT THAT EXCEPTION HANDLERS (ON TASK BODIES AND ELSEWHERE) ONLY APPLY TO EXCEPTIONS RAISED IN THE CORRESPONDING STATEMENT SEQUENCE. EXCEPTIONS RAISED IN THE CORRESPONDING DECLARATIVE PART ARE TREATED AS UNHANDLED EXCEPTIONS. (THIS RULE ENSURES THAT THE DECLARATIVE PART HAS BEEN COMPLETELY ELABORATED WHEN THE HANDLER IS INVOKED, SO THAT THE HANDLER CAN REFER TO ANY ENTITY DECLARED THERE.)

THE DECLARATIVE PART OF Hopeless_Task_Type'S TASK BODY ALWAYS RAISES Constraint_Error.

IN THE FIRST BLOCK STATEMENT, EACH COMPONENT OF Hopeless_Task_List COMPLETES (AND ANY CORRECT TASK DECLARED IN THE SAME PLACE FINISHES ELABORATING ITS DECLARATIVE PART). THEN, AT THE BEGINNING OF THE BLOCK'S STATEMENT SEQUENCE, Tasking_Error IS RAISED (ONCE, EVEN THOUGH TEN TASKS FAILED). AFTER THE EXCEPTION IS HANDLED, ANY CORRECT TASKS DECLARED IN THE BLOCK STATEMENT MUST TERMINATE BEFORE THE BLOCK STATEMENT IS LEFT. (SLIDE 17-6 ADDRESSES THIS POINT.)

IN THE SECOND BLOCK STATEMENT, THE ALLOCATOR EVALUATES TO A POINTER TO TWENTY COMPLETED TASKS, AND THEN Tasking_Error IS RAISED BEFORE THE ASSIGNMENT TO Hopeless_Task_List_Pointer. THE HANDLER IS INVOKED ONCE.

SINCE EACH HANDLER IS INVOKED ONCE, THE VALUE ASSIGNED TO Exception_Count IS 2.

EXCEPTIONS IN TASK BODY DECLARATIVE PARTS: EXAMPLE

```
procedure Example (Exception_Count : out Natural) is
   task type Hopeless_Task_Type;
   type Hopeless_Task_List_Type is array (Positive range <>) of Hopeless_Task_Type;
   Number_Of_Exceptions : Natural := 0;

   task body Hopeless_Task_Type is
      Alpha : array (1 .. 10) of Integer;
      Beta  : Integer := Alpha (11); -- Raises Constraint_Error, completing task
   begin
      -- This sequence of statements is never reached.
      ...
   exception
      -- This handler does not apply to the declarative part and is not invoked;
      when Constraint_Error => Alpha := (Alpha'Range => 0);
   end Hopeless_Task_Type;

begin -- Example
   declare
      Hopeless_Task_List : Hopeless_Task_List_Type (1 .. 10);
   begin
      -- Tasking_Error raised here.
      ...
   exception
      when Tasking_Error => Number_Of_Exceptions := Number_Of_Exceptions + 1;
   end;
   declare
      type Hopeless_Task_List_Pointer_Type is access Hopeless_Task_List_Type;
      Hopeless_Task_List_Pointer : Hopeless_Task_List_Pointer_Type;
   begin
      Hopeless_Task_List_Pointer := new Hopeless_Task_List_Type (1 .. 20);
      -- Tasking_Error raised by allocator.
   exception
      when Tasking_Error => Number_Of_Exceptions := Number_Of_Exceptions + 1;
   end;
   Exception_Count := Number_Of_Exceptions;
end Example;
```

- BULLET 3:

  THIS IS IN CONTRACT TO THE RULES FOR SUBPROGRAMS AND BLOCK STATEMENTS. AN EXCEPTION NOT HANDLED BY A SUBPROGRAM IS RE-RAISED AT THE SUBPROGRAM CALL. AN EXCEPTION NOT HANDLED BY A BLOCK STATEMENT IS RE-RAISED IN THE STATEMENT SEQUENCE CONTAINING THE BLOCK SEQUENCE.

  AN UNHANDLED EXCEPTION IN ONE TASK COMPLETES THAT TASK, BUT DOES NOT DIRECTLY AFFECT OTHER TASKS.

EXCEPTIONS IN TASK BODY STATEMENT SEQUENCES

- IF EXCEPTION IS HANDLED IN A BLOCK STATEMENT, NORMAL EXECUTION RESUMES AT THE POINT FOLLOWING THE BLOCK STATEMENT.

```
task body Example_1 is
   ...
begin
      ...
      begin
         A := B*C;
      exception
         when Numeric_Error => A := Float'Last;
      end;
      ...
end Example_1;
```

- IF THE EXCEPTION IS HANDLED BY THE TASK BODY'S EXCEPTION HANDLER, THE TASK BECOMES COMPLETED AFTER EXECUTION OF THE HANDLER.

```
task body Example_2 is
   ...
begin
   ...
exception
   when others => Clean_Up;
end Example_2;
```

- IF THE EXCEPTION IS NOT HANDLED, THE TASK BECOMES COMPLETED, BUT THE EXCEPTION IS NOT PROPAGATED. NO EXCEPTION IS RAISED ELSEWHERE.

THIS IS ONE PARTICULAR WAY THAT EXCEPTION CAN BE RAISED IN A TASK BODY STATEMENT SEQUENCE.

- BULLET 2:

  THIS IS A RULE OF STYLE. THE I/O PACKAGES RAISE EXCEPTIONS TO INDICATE SITUATIONS THAT CANNOT BE ANTICIPATED OR DETECTED IN ANY OTHER WAY (E.G. NON-EXISTENT FILES, UNREADABLE TAPES); IT IS USUALLY SIMPLER AND CLEARER TO HANDLE Numeric_Error THAN TO DETERMINE BEFORE HAND WHETHER A GIVEN ARITHMETIC EXPRESSION WILL OVERFLOW; BUT Program_Error IS MEANT TO BE INDICATING OF AN IMPROPERLY WRITTEN PROGRAM.

AN EXCEPTION RAISED BY SELECTIVE WAITS

- A SELECTIVE WAIT WITH A GUARD ON EACH ALTERNATIVE AND NO else PART RAISES Program_Error WHEN ALL GUARDS EVALUATE TO False.

```
select
   when Current_Count > 0 =>
      accept Decrement;
      Current_Count := Current_Count - 1;
or
   when Current_Count < 0 =>
      accept Increment;
      Current_Count := Current_Count + 1;
end select;
-- Raises Program_Error if Current_Count is zero.
```

- PROGRAMMERS SHOULD TRY TO PREVENT Program_Error FROM EVER BEING RAISED.

- THE RULES GIVEN ON THE PREVIOUS SLIDE (FOR EXCEPTIONS RAISED IN A TASK BODY'S SEQUENCE OF STATEMENTS) APPLY.

INSTRUCTOR NOTES

- BULLET 1;

  THIS IS JUST A CONSEQUENCE OF RULES FOR TERMINATION THAT WERE GIVEN IN SECTION 6.

- BULLET 2:
  - ITEM 1: SEE Exception_1 ABOVE.
  - ITEM 2: SEE Exception_2 ABOVE.

EXCEPTIONS IN MASTERS

- IF THE MASTER OF A TASK RAISES AN EXCEPTION (PERHAPS FOR SOME REASON UNRELATED TO TASKING), THE MASTER IS NOT LEFT UNTIL ALL ITS DEPENDENTS HAVE TERMINATED.

```
procedure Master is
   task Dependent_Task;
   task body Dependent_Task is
      ...
   begin
      ...
   end Dependent_Task;
begin -- Master
   ...
   if ... then
      ...
      raise Exception_1;  -- Handled
   elsif ... then
      ...
      raise Exception_2;  -- Propagated
   else
      ...
   end if;
   ...
exception
   when Exception_1 =>
      ...
end Master;
```

- TWO CASES TO CONSIDER:
  - IF THE MASTER HAS A HANDLER FOR THE EXCEPTION, IT IS INVOKED IMMEDIATELY. FOLLOWING EXECUTION OF THE HANDLER, DEPARTURE FROM THE MASTER AWAITS TERMINATION OF ITS DEPENDENT TASKS.
  - IF THERE IS NO HANDLER, PROPAGATION DOES NOT OCCUR UNTIL ALL DEPENDENT TASKS HAVE TERMINATED.

INSTRUCTOR NOTES

- BULLET 1:

  THE EXCEPTION IS SIMULTANEOUSLY PROPAGATED TO TWO DIFFERENT TASKS. EACH RE-RAISING OF THE EXCEPTION IS INDEPENDENT OF THE OTHER. EACH TASK MAY HANDLE ITS OWN RE-RAISING OF THE EXCEPTION. IF ONLY ONE TASK HANDLES IT, IT REMAINS UNHANDLED IN THE OTHER TASK.

- BULLET 2:

  THIS EXAMPLE IS TAKEN FROM THE ELEVATOR EXERCISE IN SECTION 12. TO GUARD AGAINST THE POSSIBILITY OF A PROGRAMMING ERROR IN THE TASK MONITORING CALL BUTTONS, THE Report_Request_Mode ENTRY VALIDATES ITS PARAMETERS. AN ENTRY CALL WITH INVALID PARAMETERS (SPECIFYING THAT THE "Down" BUTTON OF THE BOTTOM FLOOR OR THE "Up" BUTTON OF THE TOP FLOOR WAS PRESSED) RAISES Invalid_Request_Report.

  THIS BLOCK STATEMENT HANDLES THE EXCEPTION WITHIN THE CALLED TASK BY IGNORING IT. THE NULL HANDLER KEEPS THE EXCEPTION FROM BEING PROPAGATED, SO THAT CONTROL STAYS WITHIN THE LOOP.

  THE BLOCK STATEMENT CANNOT BE PLACED DIRECTLY AROUND THE accept STATEMENT BECAUSE AN ALTERNATIVE OF A SELECTIVE WAIT CANNOT BEGIN WITH A BLOCK STATEMENT.

EXCEPTIONS ARISING IN accept STATEMENTS

- IF AN EXCEPTION IS RAISED INSIDE AN accept STATEMENT:
  - EXECUTION OF THE REMAINDER OF THE accept STATEMENT IS ABANDONED.
  - WITHIN THE CALLING TASK, THE EXCEPTION IS RE-RAISED BY THE ENTRY CALL STATEMENT.
  - WITHIN THE CALLED TASK, THE EXCEPTION IS RE-RAISED BY THE accept STATEMENT ITSELF.
- USING EXCEPTIONS TO "IDIOT-PROOF" ENTRY PARAMETERS:
  - THE accept STATEMENT CHECKS PARAMETER VALUES AND RAISES AN EXCEPTION.
  - THE EXCEPTION IS PROPAGATED TO THE ENTRY CALL.
  - WITHIN THE BODY OF THE CALLED TASK, THE EXCEPTION IS HANDLED BY AN OUTER BLOCK STATEMENT, SO THAT THE TASK REMAINS ALIVE TO SERVICE LATER ENTRY CALLS.

```
loop
   begin
      select
         accept Report_Request_Made (Floor : in Floor_Type;
                                     Direction : in Direction_Type) do
            if (Floor = Floor_Type'First and Direction = Down) or
               (Floor = Floor_Type'Last and Direction = Up) then
               raise Invalid_Request_Report;  -- Here and in calling task
            elsif Status_Table (Floor, Direction) = Absent then
               Status_Table (Floor, Direction) := Pending;
            end if;
         end Report_Request_Made;
      or
         ...
      or
         terminate;
      end select;
   exception
      when Invalid_Request_Report =>
         null;
   end; -- block statement
end loop;
```

INSTRUCTOR NOTES

- BULLET 2:

  BEFORE THE RENDEZVOUS BEGINS, THE TASKS ARE ASYNCHRONOUS, SO THE CALLING TASK MAY ASSUME NOTHING ABOUT THE CURRENT PROGRESS OF THE CALLED TASK. ALL IT "KNOWS" IS THAT THE TASK TO BE CALLED BECAME COMPLETED OR ABNORMAL BEFORE ACCEPTING THE ENTRY CALL (WHETHER THIS HAPPENED BEFORE OR AFTER THE CALL WAS ISSUED).

- BULLET 3:

  - ITEM 2: THE if STATEMENT IS NOT FOOLPROOF, BUT IT DOES GREATLY REDUCE THE PROBABILITY THAT Tasking_Error WILL BE RAISED.

CALLING A COMPLETED OR ABNORMAL TASK

- Tasking_Error IS RAISED BY AN ENTRY CALL WHEN ...
  - THE CALLED TASK IS ALREADY COMPLETED, OR HAS BEEN RENDERED ABNORMAL BY AN abort STATEMENT.
  - THE CALLED TASK BECOMES COMPLETED, OR IS RENDERED ABNORMAL BY AN abort STATEMENT AFTER ITS ENTRY HAS BEEN CALLED BUT BEFORE IT HAS ACCEPTED THE CALL.
- FROM THE PERSPECTIVE OF THE CALLING TASK, THESE SITUATIONS ARE INDISTINGUISHABLE.
- IF T IS A TASK OBJECT, THE ATTRIBUTE T'Callable RETURNS A Boolean VALUE INDICATING WHETHER T's ENTRIES CAN BE CALLED WITHOUT RAISING Tasking_Error.
  - False WHEN T IS ABNORMAL, COMPLETED, OR TERMINATED, True OTHERWISE.
  - CAN CHANGE FROM True to False BETWEEN THE TIME IT IS TESTED AND THE TIME AN ENTRY CALL STATEMENT IS EXECUTED.

```
if T'Callable then
   -- T now becomes uncallable
   T.Some_Entry;  -- Raises Tasking_Error despite precaution
else
   ...
end if;
```

INSTRUCTOR NOTES

THIS ASYMMETRY REFLECTS THE ROLES OF SERVER AND USER TASKS EXPLAINED IN SECTION 11.

- BULLET 1:

  THIS IS A GENERALIZATION OF THE RULE ON THE PREVIOUS SLIDE. Tasking_Error IS RAISED NOT ONLY IF THE CALLED TASK BECOMES ABNORMAL BEFORE REACHING AN accept STATEMENT, BUT IF IT BECOMES ABNORMAL BEFORE COMPLETING AN accept STATEMENT.

- BULLET 2:

  THIS IS A REITERATION OF BULLET 3 ON SLIDE 16-4.

TASKS ABORTED WHILE COMMUNICATING

- IF A CALLED TASK IS ABORTED DURING A RENDEZVOUS, Tasking_Error IS RAISED IN THE CALLING TASK AT THE POINT OF THE ENTRY CALL.

- IF A CALLING TASK IS ABORTED AFTER ISSUING AN ENTRY CALL, THE CALLED TASK NEVER KNOWS THE DIFFERENCE.
  - IF THE CALL HAS NOT YET BEEN ACCEPTED, IT IS CANCELLED (REMOVED FROM THE ENTRY QUEUE).
  - IF THE RENDEZVOUS HAS STARTED, IT FINISHES NORMALLY, AND THE CALLED TASK BECOMES COMPLETED ONLY AFTER THE ENTRY CALL STATEMENT IS DONE.
  - EITHER WAY, THE CALLED TASK REMAINS ALIVE TO SERVE OTHER USERS.

INSTRUCTOR NOTES

ALLOW 30 MINUTES FOR THIS SECTION AND INTRODUCE THE EXERCISE AT THE END OF THE SECTION BEFORE BREAKING FOR LUNCH.

WHEN CLASS RESUMES, ALLOW 65 MINUTES FOR SOLUTION AND DISCUSSION OF THE EXERCISE.

Section 18

INTERRUPT ENTRIES

INSTRUCTOR NOTES

- BULLET 3:

  ESSENTIALLY, THE OUTSIDE WORLD IS VIEWED AS A COLLECTION OF TASKS THAT CALL ENTRIES WHEN IT IS TIME TO CAUSE AN INTERRUPT.

- BULLET 4:

  CAVEAT: THIS IS ONLY ONE POSSIBLE IMPLEMENTATION. WE DEPART FROM OUR USUAL POLICY OF NOT DESCRIBING POSSIBLE IMPLEMENTATIONS TO PERSUADE THE STUDENT THAT THIS MECHANISM IS CAPABLE OF HANDLING INTERRUPTS PROMPTLY.

  AN INTERRUPT VECTOR IS A LIST OF ADDRESSES OF THE MACHINE LANGUAGE ROUTINES THAT ARE TO BE GIVEN CONTROL BY THE HARDWARE WHEN INTERRUPTS OCCUR. THE MACHINE ARCHITECTURE ASSIGNS DIFFERENT OFFSETS WITHIN THE VECTOR TO HOLD THE ADDRESSES OF THE ROUTINES FOR DIFFERENT KINDS OF INTERRUPTS.

HANDLING HARDWARE INTERRUPTS

- HARDWARE INTERRUPTS CAN BE HANDLED BY Ada TASKS.
- AN ENTRY OF A TASK CAN BE DESIGNATED TO BE AN INTERRUPT ENTRY.
  - WHEN THE SPECIFIED INTERRUPT OCCURS, THE DESIGNATED ENTRY IS CALLED.
  - THE INTERRUPT IS HANDLED BY ACCEPTING A CALL ON THAT ENTRY.
- THIS PROVIDES A HIGH-LEVEL VIEW OF INTERRUPTS.
  - CONSISTENT WITH USUAL MEANS OF TASK SYNCHRONIZATION AND COMMUNICATION.
  - ALLOWS DEVICE DRIVERS TO BE WRITTEN AT A HIGH LEVEL OF ABSTRACTION.
- HANDLING INTERRUPTS IN THIS WAY CAN BE QUITE EFFICIENT.
  - ONE POSSIBLE IMPLEMENTATION:
    - WHEN A TASK IS READY TO ACCEPT A "CALL" ON THE INTERRUPT ENTRY, IT STORES THE ADDRESS OF THE accept STATEMENT'S OBJECT CODE IN THE HARDWARE INTERRUPT VECTOR FOR THE DESIGNATED INTERRUPT.
    - UPON AN INTERRUPT, THE HARDWARE BRANCHES DIRECTLY TO THE accept STATEMENT.

INSTRUCTOR NOTES

THIS SLIDE IS A REVIEW OF MATERIAL FROM MODULE L305. ADDRESS CLAUSES ARE USED TO SPECIFY INTERRUPT ENTRIES.

- BULLET 1:

  A THIRD USE OF ADDRESS CLAUSES, DEALING WITH THE TOPIC OF THIS SECTION, IS GIVEN ON THE NEXT SLIDE.

  ADDRESS CLAUSES SHOULD NOT BE USED TO ACHIEVE OVERLAYS.

- BULLET 2:

  THE EXPRESSION NEED NOT BE STATIC. FOR EXAMPLE, IT CAN BE A GENERIC FORMAL PARAMETER.

- BULLET 4:

  THE EXAMPLES IN THIS SECTION ASSUME THE FIRST IMPLEMENTATION.

REVIEW: ADDRESS CLAUSES

- THE ADDRESS CLAUSE IS A LOW-LEVEL FEATURE THAT ALLOWS THE PROGRAMMER TO SPECIFY THE ADDRESS OF AN ENTITY.
  - AN OBJECT (STARTING ADDRESS OF DATA)
  - A PROGRAM UNIT (ADDRESS OF FIRST MACHINE INSTRUCTION).
- FORM:

  for [entity name] use at [expression of type System.Address];

- THE PREDEFINED PACKAGE System DECLARES IMPLEMENTATION-DEPENDENT TYPES, SUBTYPES, AND NAMED NUMBERS, INCLUDING THE TYPE Address.
- SOME POSSIBLE IMPLEMENTATIONS OF System.Address (DEPENDING ON THE UNDERLYING ARCHITECTURE):
  - A NONNEGATIVE NUMBER
  - A "SEGMENT NAME" PLUS AN OFFSET WITHIN A SEGMENT
  - (IN A DISTRIBUTED SYSTEM) A "SITE NAME" PLUS AN ADDRESS INTO A PARTICULAR SITE'S STORAGE.

INSTRUCTOR NOTES

- BULLET 1:

  - ITEM 3: THIS APPLIES TO ANY USE OF AN ADDRESS CLAUSE.

- BULLET 2:

  16#40# IS A BASED INTEGER LITERAL, MEANING "40 HEXADECIMAL".

DESIGNATING INTERRUPT ENTRIES

- AN ENTRY MAY BE DESIGNATED AS AN INTERRUPT ENTRY BY AN ADDRESS CLAUSE FOR THE ENTRY.
  - THE SPECIFIED ADDRESS REFERS TO A PARTICULAR INTERRUPT, ACCORDING TO IMPLEMENTATION-DEFINED CONVENTIONS (TYPICALLY THE ADDRESS OF THE CORRESPONDING HARDWARE INTERRUPT VECTOR).
  - THE ADDRESS CLAUSE APPEARS IN THE TASK DECLARATION, SOMEPLACE AFTER THE ENTRY DECLARATION.
  - THE TASK DECLARATION MUST BE IN THE SCOPE OF A with CLAUSE FOR THE PACKAGE System.
- EXAMPLE:

```
with System;

package Serial_Interface_Package is

   task Serial_Interface_Task is
       entry Send_Character (C : in Character); -- ordinary entry
       entry Serial_Interface_Interrupt;        -- interrupt entry
       for Serial_Interface_Interrupt use at 16#40#;
   end Serial_Interface_Task;

   Serial_Interface_Malfunction:  exception;

end Serial_Interface_Package;
```

INSTRUCTOR NOTES

- BULLET 1:

  WE SHALL RETURN TO THIS POINT IN SECTION 20, WHEN DISCUSSING PRIORITIES.

- BULLET 4:

  AN ENTRY IS A PROPERTY OF A TASK OBJECT, NOT A TASK TYPE. GIVEN THE DECLARATIONS

```
task type T is
   entry E;
end T;
Obj_1, Obj_2: T;
```

  Obj_1.E AND Obj_2.E ARE TWO DISTINCT ENTRIES. IT IS ERRONEOUS FOR THE SAME HARDWARE INTERRUPT TO CORRESPOND TO BOTH THESE ENTRIES.

  - ITEM 2: IF ONLY ONE OBJECT IS TO BE DECLARED, A SINGLE TASK DECLARATION (WITHOUT THE WORD type) IS SIMPLER. THUS A GOOD GUIDELINE IS TO DECLARE INTERRUPT ENTRIES ONLY IN SINGLE TASK DECLARATIONS, NOT TASK TYPE DECLARATIONS.

PROPERTIES OF INTERRUPT ENTRIES

- IF A TASK IS WAITING AT AN accept STATEMENT FOR AN INTERRUPT ENTRY WHEN THE CORRESPONDING HARDWARE INTERRUPT OCCURS, THE accept STATEMENT BEGINS EXECUTING IMMEDIATELY, EXCEPT PERHAPS IF AN accept STATEMENT FOR ANOTHER INTERRUPT ENTRY IS ALREADY IN PROGRESS.

- AN INTERRUPT ENTRY CAN BE CALLED BY SOFTWARE AS IF IT WERE AN ORDINARY ENTRY.
  - EFFECT IS TO SIMULATE AN INTERRUPT
  - THE accept STATEMENT MIGHT NOT BE EXECUTED IMMEDIATELY.

- AN IMPLEMENTATION MAY ALLOW/REQUIRE ENTRY PARAMETERS FOR INTERRUPT ENTRIES CORRESPONDING TO CERTAIN HARDWARE INTERRUPTS.
  - PARAMETERS MUST BE OF MODE in.
  - USED TO CONVEY CONTROL INFORMATION (E.G. PROGRAM STATUS WORD, INTERRUPT CODE).

- THE SAME HARDWARE INTERRUPT SHOULD NOT CORRESPOND TO MORE THAN ONE ENTRY (OF ONE TASK OBJECT).
  - CONTROL MUST PASS TO A UNIQUELY DEFINED PLACE WHEN AN INTERRUPT OCCURS.
  - THIS MAKES INTERRUPT ENTRIES INAPPROPRIATE IN TASK TYPE DECLARATIONS (UNLESS ONLY ONE TASK OBJECT OF THAT TYPE IS TO BE DECLARED).

INSTRUCTOR NOTES

- BULLET 1:

  A SERIAL INTERFACE IS A DEVICE FOR TRANSMITTING (OR RECEIVING) A BYTE ONE BIT AT A TIME.

- BULLET 2:

  THE PACKAGE Low_Level_IO WAS DISCUSSED IN MODULE L305. EACH IMPLEMENTATION HAS ITS OWN VERSION, BUT EACH VERSION HAS (AMONG OTHER THINGS) ONE OR MORE TYPES LIKE Port_Type FOR NAMING DEVICES AND ONE OR MORE OVERLOADED VERSIONS OF Send_Central.

EXAMPLE: A TASK FOR TRANSMITTING CHARACTERS

- EXTERNAL INTERFACE:
  - A CALL ON Serial_Interface_Task.Send_Character TRANSMITS A SPECIFIED CHARACTER THROUGH A SERIAL INTERFACE.
  - IF THE SERIAL INTERFACE MALFUNCTIONS WHILE ATTEMPTING TO TRANSMIT A CHARACTER, THE ENTRY CALL SHOULD RAISE THE EXCEPTION Serial_Interface_Malfunction.
- ASSUMPTIONS:
  - THE FOLLOWING DECLARATIONS (AMONG OTHERS) ARE PROVIDED BY THE PREDEFINED PACKAGE Low_Level_IO:

    ```
    type Port_Type is range 0 .. 255;
    procedure Send_Control (Device : in Port_Type; Data : in out Character);
    ```

  - A Send_Control TO PORT 16#10# CAUSES THE SERIAL INTERFACE TO START TRANSMITTING THE CHARACTER SPECIFIED BY DATA.
  - NORMALLY, THE SERIAL INTERFACE RESPONDS WITH AN INTERRUPT WHEN IT IS READY FOR ANOTHER CHARACTER. THE VECTOR FOR THIS INTERRUPT IS AT ADDRESS 16#40#.
  - IF THIS INTERRUPT DOES NOT ARRIVE WITHIN 0.01 SECONDS, THE SERIAL INTERFACE IS MALFUNCTIONING.

THE TASK REPEATEDLY EXECUTES A BLOCK STATEMENT.

THE BLOCK STATEMENT WAITS FOR A CALL ON THE ORDINARY ENTRY Send_Control. WHEN THE CALL IS RECEIVED, THE accept STATEMENT PERFORMS THE OUTPUT OPERATION AND THEN EXECUTES A NESTED SELECTIVE WAIT. THIS SELECTIVE WAIT IS FOR EITHER A "CALL" ON THE INTERRUPT ENTRY Serial_Interface_Interrupt OR THE EXPIRATION OF THE 0.01 SECOND DELAY, WHICHEVER COMES FIRST. IF THE DELAY EXPIRES FIRST, Serial_Interface_Malfunction IS RAISED.

BECAUSE IT IS RAISED FROM WITHIN THE accept STATEMENT FOR Send_Character, THE EXCEPTION IS PROPAGATED TO THE CALL ON THAT ENTRY. THE EXCEPTION IS ALSO RE-RAISED WITHIN THE BLOCK STATEMENT OF Serial_Interface_Task, WHERE IT IS HANDLED BY A NULL HANDLER. (THIS IS JUST LIKE THE Invalid_Request_Report EXAMPLE ON SLIDE 17-7.) THIS KEEPS Serial_Interface_Task ALIVE TO HANDLE OTHER CALLS ON Send_Character. IT MAY BE, FOR EXAMPLE, THAT THE CALLING TASK RESPONDS TO A Serial_Interface_Malfunction EXCEPTION BY RETRYING THE CALL ON Send_Character IN THE HOPE THAT THE MALFUNCTION WAS TRANSIENT.

NORMALLY, accept STATEMENTS SHOULD BE KEPT AS SHORT AS POSSIBLE TO AVOID BLOCKING THE CALLING TASK. THE accept STATEMENT FOR Send_Character VIOLATES THIS GUIDELINE. IT CONTAINS A SELECTIVE WAIT NESTED INSIDE IT, SO THAT THE EXCEPTION CAN BE RAISED INSIDE THE accept STATEMENT AND PROPAGATED TO THE ENTRY CALL. SINCE THE ENTRY CALL IS TO EITHER RAISE OR NOT RAISE AN EXCEPTION, DEPENDING ON WHETHER AN INTERRUPT ARRIVES WITHIN 0.01 SECONDS, A POTENTIALLY LONG ENTRY CALL IS INHERENT IN THE PROBLEM SPECIFICATION. THIS DOES NOT REALLY SLOW DOWN THE SYSTEM, CHARACTERS CANNOT BE TRANSMITTED FASTER THAN THE SERIAL INTERFACE ISSUES INTERRUPTS, AND Send_Character CANNOT BE CALLED MORE FREQUENTLY THAN CHARACTERS ARE TRANSMITTED. IT ALL COMES DOWN TO WHETHER THE CALLING TASK WAITS BEFORE OR DURING A RENDEZVOUS.

```
with System;
package Serial_Interface_Package is
   task Serial_Interface_Task is
      entry Send_Character (C : in Character);
      entry Serial_Interface_Interrupt;
      for Serial_Interface_Interrupt use at 16#40#;
   end Serial_Interface_Task;

   Serial_Interface_Malfunction:  exception;

end Serial_Interface_Package;

with Low_Level_IO;
package body Serial_Interface_Package is
   task body Serial_Interface_Task is
      Serial_Interface_Port : constant := 16#10#;
      Interrupt_Time_Limit : constant := 0.01;
   begin
      loop
         begin
             accept Send_Character (C : in Character) do
                Low_Level_IO.Send_Control (Serial_Interface_Port, C);
                select
                    accept Serial_Interface_Interrupt;
                or
                    delay Interrupt_Time_Limit;
                    raise Serial_Interface_Malfunction;
                end select;
             end Send_Character;
         exception
             when Serial_Interface_Malfunction =>
                null;
         end;
      end loop;
   end Serial_Interface_Task;
end Serial_Interface_Package;
```

INSTRUCTOR NOTES

- BULLET 2:

  - ITEM 2: THUS THE CHARACTER TYPED WILL BE PASSED AS AN in PARAMETER OF THE INTERRUPT ENTRY.

- BULLET 3;

  WE OMIT CODE FOR ECHOING CHARACTERS, BACKING UP THE CURSOR AND ERASING CHARACTERS UPON BACKSPACE AND CONTROL-X, ETC.

EXAMPLE: KEYBOARD INPUT HANDLER

- EXTERNAL INTERFACE:
  - PACKAGE SPECIFICATION:

```
package Keyboard_Package is
    Input_Buffer_Size : constant := 80;
    subtype Input_Buffer_Subtype is String (1 .. Input_Buffer_Size);
    subtype Line_Length_Subtype is Integer range 0 .. Input_Buffer_Size;
    procedure Get_Line
       (Input_Buffer : out Input_Buffer_Subtype;
        Line_Length : out Line_Length_Subtype);
end Keyboard_Package;
```

  - LINES ARE TYPED IN A CHARACTER AT A TIME, ENDING WITH A CARRIAGE RETURN (ASCII.CR).
  - CONTROL-H (ASCII.BS) CANCELS THE PREVIOUSLY TYPED CHARACTER (EXCEPT AT THE BEGINNING OF THE LINE).
  - CONTROL-X (ASCII.CAN) CANCELS ALL CHARACTERS THAT HAVE BEEN TYPED SO FAR ON THE CURRENT LINE.
  - IF THE CURRENT LINE ALREADY CONTAINS 80 CHARACTERS, ALL KEYSTROKES OTHER THAN CARRIAGE RETURN, CONTROL-H, AND CONTROL-X ARE IGNORED.
  - A CALL ON Get_Line PLACES THE CONTENTS OF THE MOST RECENTLY TYPED LINE (NOT INCLUDING THE CARRIAGE RETURN) IN Input_Buffer, AND THE LENGTH OF THE LINE IN Line_Length.
- ASSUMPTIONS:
  - A KEYSTROKE GENERATES AN INTERRUPT TO VECTOR 16#30#.
  - THE CHARACTER ENTERED IS PROVIDED AS CONTROL INFORMATION BY THE INTERRUPT.
- SIMPLIFICATION:
  - THE LINE BEING ENTERED NEED NOT BE DISPLAYED.

INSTRUCTOR NOTES

THE Keyboard_Handler TASK BODY WILL BE WRITTEN AS IF IT IS SERVIN TWO TASKS - ONE THAT CALLS Keystroke_Interrupt TO DELIVER CHARACTERS AND ONE THAT CALLS Assemble_Line TO OBTAIN AN ASSEMBLED LINE.

TO A USER OF Keyboard_Package, THE Keystroke_Interrupt ENTRY IS PART OF THE IMPLEMENTATION. AS FAR AS THE USER IS CONCERNED, THE ONLY SERVICE PROVIDED BY Keyboard_Package IS TO DELIVER ASSEMBLED LINES. THE PACKAGE HIDES THE INTERRUPT ENTRY FROM THE USER.

Keyboard_Package BODY

```
with System;

package body Keyboard_Package is

   Keystroke_Interrupt_Address : constant := 16#30#;

   task Keyboard_Handler is
      entry Keystroke_Interrupt (C : in Character);
      entry Assemble_Line
        (Input_Buffer : out Input_Buffer_Subtype; Line_Length : out Line_Length_Subtype);
      for Keystroke_Interrupt use at Keystroke_Interrupt_Address;
   end Keyboard_Handler;

   task body Keyboard_Handler is separate; -- Subunit on next slide

   procedure Get_Line
      (Input_Buffer : out Input_Buffer_Subtype; Line_Length : out Line_Length_Subtype) is
   begin
      Keyboard_Handler.Assemble_Line (Input_Buffer, Line_Length);
   end Get_Line;

end Keyboard_Package;
```

INSTRUCTOR NOTES

WITHIN THE TASK BODY, THE INTERRUPT ENTRY IS TREATED JUST LIKE ANY OTHER ENTRY.

THE OUTER LOOP REPEATEDLY CLEARS THE BUFFER, EXECUTES THE INNER LOOP TO RECEIVE AND ACT UPON KEYSTROKES UNTIL A CARRIAGE RETURN IS ENCOUNTERED, AND ACCEPTS A CALL ON Assemble_Line TO DELIVER THE ASSEMBLED LINE.

Keyboard_Package.Get_Line MIGHT BE CALLED BY ANOTHER TASK THAT BUFFERS COMPLETE INPUT LINES UNTIL THE PROGRAM IS READY TO EXAMINE THEM. THIS WOULD ALLOW THE OPERATOR TO "TYPE AHEAD" OF THE PROGRAM.

Keyboard_Handler SUBUNIT

```
separate (Keyboard_Package)
task body Keyboard_Handler is
      Buffer         : Input_Buffer_Subtype;
      Current_Length : Input_Length_Subtype;
      Keystroke      : Character;
begin
   loop -- one repetition for each line
      Current_Length := 0;
      loop -- one repetition for each Keystroke in the line
         accept Keystroke_Interrupt (C : in Character) do
             Keystroke := C;
         end Keystroke_Interrupt;
         exit when Keystroke = ASCII.CR;
         case Keystroke is
             when ASCII.BS =>
                if Current_Length > 0 then
                    Current_Length := Current_Length - 1;
                end if;
             when ASCII.CAN =>
                Current_Length := 0;
             when others =>
                if Current_Length < 80 then
                    Current_Length := Current_Length + 1;
                    Buffer (Current_Length) := Keystroke;
                end if;
         end case;
      end loop;
      accept Assemble_Line (Input_Buffer : out Input_Buffer_Subtype;
                            Line_Length  : out Line_Length_Subtype) do
         Input_Buffer := Buffer;
         Line_Length := Current_Length;
      end Assemble_Line;
   end loop;
end Keyboard_Handler;
```

INSTRUCTOR NOTES

THE 65 MINUTES ALLOCATED FOR THIS EXERCISE SHOULD ALLOW A COMPLETE SOLUTION. THE SOLUTION PUTS A SIMPLE MONITOR THE STUDENTS HAVE ALREADY SEEN INTO A MORE COMPLETE CONTEXT.

THE INTERRUPT IS GENERATED WHENEVER ANY CYLINDER IN THE ENGINE FIRES. THIS IS HANDLED BY THE HARDWARE. THE PROGRAM DOES NOT KNOW OR CARE ABOUT THE ACTUAL NUMBER OF CYLINDERS.

THE ASSUMPTIONS ABOUT System.Address ARE THOSE WE USED IN THIS SECTION.

THE MONITOR IN THE SOLUTION DOES NOT HAVE A terminate ALTERNATIVE BECAUSE IT IS DECLARED IN A LIBRARY PACKAGE AND RUNS FOR AS LONG AS THE MICROPROCESSOR RUNNING IT STAYS ON.

EXERCISE 18.1

A PROGRAM FOR A MICROPROCESSOR - CONTROLLED INTERNAL COMBUSTION ENGINE HAS THE FOLLOWING PACKAGE:

```
package Engine_Speed_Package is
   function Number_Of_Cylinders_Fired return Natural;
end Engine_Speed_Package;
```

EACH CALL ON Number_Of_Cylinders_Fired RETURNS THE NUMBER OF CYLINDERS THAT HAVE FIRED SINCE THE PREVIOUS CALL ON THE FUNCTION.

WRITE THE Engine_Speed_Package BODY, USING THE FOLLOWING ASSUMPTIONS:

- EACH TIME A CYLINDER IS FIRED, IT GENERATES AN INTERRUPT TO VECTOR 16#50#.
- System.Address IS AN INTEGER TYPE
- INTERRUPTS ARE IDENTIFIED IN ADDRESS CLAUSES BY THEIR INTERRUPT VECTOR ADDRESSES

INSTRUCTOR NOTES

ALLOW 50 MINUTES FOR THIS SECTION. TAKE A BREAK ABOUT MIDWAY THROUGH THE SECTION. (AFTER SLIDE 19-8 IS AN APPROPRIATE PLACE.)

Section 19

ENTRY FAMILIES

INSTRUCTOR NOTES

TO MOTIVATE THE NEED FOR ENTRY FAMILIES, THIS SLIDE EXAMINES AN ANALOGOUS SITUATION HAVING NOTHING TO DO WITH TASKING.

- BULLET 1:

  THE PROGRAM *Print_Deviations* READS SIX REAL LITERALS FROM THE STANDARD INPUT FILE AND WRITES THE DEVIATION OF EACH LITERAL FROM THE MEAN ON THE STANDARD OUTPUT FILE. THE PROGRAM ON THE LEFT SHOWS HOW WE MIGHT SOLVE THIS PROBLEM IF THERE WERE NO SUCH THING AS AN ARRAY. THE PROGRAM ON THE RIGHT USES AN ARRAY.

- BULLET 2:

  - ITEM 2:

    - SUBITEM 1: RATHER THAN SPECIFYING AN INSTANCE OF THE PATTERN, AS ON THE LEFT (SIX CALLS ON Get FOLLOWED BY COMPUTATION OF THE Mean, FOLLOWED BY SIX CALLS ON Put), THE LOOPS ON THE RIGHT GIVE THE RATIONALE FOR THE SEQUENCE OF ACTIONS (ONE CALL ON Get AND ONE ADDITION FOR EACH ARRAY COMPONENT, FOLLOWED BY COMPUTATION OF THE MEAN, FOLLOWED BY ONE CALL ON Put FOR EACH ARRAY ELEMENT).

    - SUBITEM 2: BECAUSE THE PROGRAM ON THE RIGHT IS EXPRESS IN TERMS OF A PATTERN, IT IS EASY TO SPECIFY A DIFFERENT INSTANCE OF THE PATTERN. IT IS ONLY NECESSARY TO CHANGE THE UPPER BOUND IN THE DECLARATION OF A.

  - ITEM 3: THIS COULD BE PART OF A LOOP TO COUNT THE NUMBER OF TIMES EACH OF THE 128 ASCII CHARACTERS OCCURS IN SOME FILE. WITHOUT ARRAYS, WE WOULD HAVE TO MAINTAIN 128 SCALAR VARIABLES, AND USE A HUGE case STATEMENT TO DETERMINE WHICH ONE TO INCREMENT.

WHY ARE ARRAYS HELPFUL IN PROGRAMMING?

- COMPARE THESE TWO SOLUTIONS TO THE SAME PROBLEM:

```
with Text_IO; use Text_IO;
procedure Print_Deviations is
   A1, A2, A3, A4, A5, A6 : Float;
   Mean                   : Float;

   package Type_Float_IO is
      new Float_IO (Float);
   use Type_Float_IO;

begin
   Get (A1);
   Get (A2);
   Get (A3);
   Get (A4);
   Get (A5);
   Get (A6);
   Mean := (A1+A2+A3+A4+A5+A6) / 6.0;
   Put (A1 - Mean);
   Put (A2 - Mean);
   Put (A3 - Mean);
   Put (A4 - Mean);
   Put (A5 - Mean);
   Put (A6 - Mean);
end Print_Deviations;
```

```
with Text_IO; use Text_IO;
procedure Print_Deviations is
   A    : array (1 .. 6) of Float;
   Sum  : Float := 0;
   Mean : Float;

   package Type_Float_IO is
      new Float_IO (Float);
   use Type_Float_IO;

begin
   for I in A'Range loop
      Get (A(I));
      Sum := Sum + A(I);
   end loop;
   Mean := Sum / Float (A'Length);
   for I in A'Range loop
      Put (A(I) - Mean);
   end loop;

end Print_Deviations
```

- BENEFITS OF ARRAYS
  - ELIMINATION OF TEDIOUS REPETITION
  - ABSTRACT DESCRIPTION OF PATTERNS
    - HIGH-LEVEL VIEW
    - EXTENDIBLE (E.G. TO 100 NUMBERS INSTEAD OF 6)
  - RUNTIME DETERMINATION OF THE VARIABLE TO BE USED IN A GIVEN STATEMENT.

```
Frequency (Char) := Frequency (Char) + 1;
```

INSTRUCTOR NOTES

THIS COULD BE PART OF A TASK RESPONSIBLE FOR ALLOCATION OF BUFFERS. Buffer_Pool AND Available_Count TOGETHER FORM A VARIABLE-LENGTH LIST OF POINTERS TO BUFFERS. BUFFERS ARE TAKEN FROM THE RIGHT END OF THE LIST UPON ALLOCATION.

THE TASK HAS FOUR DIFFERENT ENTRIES FOR REQUESTING A BUFFER:

- Emergency_Request
- Expedited_Request
- Normal_Request
- Background_Request

THESE ENTRIES CORRESPOND TO REQUESTS OF DIFFERING PRECEDENCE. A CALL ON A GIVEN ENTRY IS NEVER TO BE ACCEPTED WHEN A CALL ON A HIGHER-PRECEDENCE ENTRY IS PENDING.

(AVOID THE TERM "PRIORITY" WHEN DISCUSSING THIS EXAMPLE. IT HAS A DIFFERENT MEANING, EXPLAINED IN SECTION 20.)

THE NESTED select STATEMENTS CONTAIN accept ALTERNATIVES THAT ARE IDENTICAL EXCEPT FOR THE ENTRY NAME. CALLS FOR EACH ENTRY ARE QUEUED INDEPENDENTLY. THE NESTING CONTROLS THE ORDER IN WHICH QUEUES ARE EXAMINED. (A CALL ON Expedited_Request IS ONLY ACCEPTED WHEN THERE ARE NO PENDING CALLS ON Emergency_Request, AND SO FORTH.)

THE NEED FOR "ARRAYS" OF ENTRIES

- PROBLEM: AS LONG AS UNALLOCATED BUFFERS REMAIN, ACCEPT ENTRY CALLS REQUESTING BUFFERS. THERE ARE FOUR ENTRIES, CORRESPONDING TO DIFFERENT PRECEDENCE LEVELS OF REQUESTS. HIGHER-PRECEDENCE CALLS SHOULD BE ACCEPTED FIRST.

- SOLUTION (REMINISCENT OF LEFT HAND PROGRAM ON PREVIOUS SLIDE):

```
while Available_Count > 0 loop
   select
      accept Emergency_Request (Buffer_Pointer : out Buffer_Pointer_Type) do
         Buffer_Pointer := Buffer_Pool (Available_Count);
      end Emergency_Request;
      Available_Count := Available_Count - 1;
   else
      select
         accept Expedited_Request (Buffer_Pointer : out Buffer_Pointer_Type) do
            Buffer_Pointer := Buffer_Pool (Available_Count);
         end Expedited_Request;
         Available_Count := Available_Count - 1;
      else
         select
            accept Normal_Request (Buffer_Pointer : out Buffer_Pointer_Type) do
               Buffer_Pointer := Buffer_Pool (Available_Count);
            end Normal_Request;
            Available_Count := Available_Count - 1;
         else
            select
               accept Background_Request (Buffer_Pointer : out Buffer_Pointer_Type) do
                  Buffer_Pointer := Buffer_Pool (Available_Count);
               end Background_Request;
               Available_Count := Available_Count - 1;
            else
               exit;
            end select;
         end select;
      end select;
   end select;
end loop;
```

INSTRUCTOR NOTES

THE TITLE AND BULLET 3 USE THE WORD "ARRAY" LOOSELY. TECHNICALLY, AN ARRAY IS A DATA OBJECT WHOSE COMPONENTS ARE DATA OBJECTS.

- BULLET 4: THESE ARE THE BENEFITS OF ARRAYS LISTED AT THE BOTTOM OF SLIDE 19-1.

ENTRY FAMILIES

- A TASK OBJECT CAN HAVE A FAMILY OF ENTRIES WITH IDENTICAL FORMAL PARAMETERS.

- THE MEMBERS OF THE FAMILY ARE DISTINGUISHED USING AN INDEX COMPUTED AT RUNTIME.
  - AN ENTRY CALL CAN CALL THE nth MEMBER OF AN ENTRY FAMILY.
  - AN accept STATEMENT CAN WAIT FOR A CALL ON THE nth MEMBER OF AN ENTRY FAMILY.
  - n CAN BE SPECIFIED BY AN ARBITRARY EXPRESSION OF THE APPROPRIATE TYPE.

- A FAMILY OF ENTRIES IS LIKE A ONE-DIMENSIONAL "ARRAY" OF ENTRIES.

- ENTRY FAMILIES FACILITATE DESCRIPTION OF TASK INTERACTIONS INVOLVING A NUMBER OF SIMILAR ENTRIES.
  - WITHOUT TEDIOUS REPETITION
  - EXPRESSED IN TERMS OF PATTERNS
  - WITH RUNTIME IDENTIFICATION OF THE ENTRY TO BE CALLED OR ACCEPTED.

THE DECLARATION OF Obtain_Buffer INDICATES THAT Obtain_Buffer IS A FAMILY OF ENTRIES INDEXED BY Precedence_Type VALUES. EACH ENTRY IN THIS FAMILY HAS AN out PARAMETER NAMED Buffer_Pointer, OF TYPE Buffer_Pointer_Type.

- BULLET 3:

  THE for LOOP PERFORMS THE SAME FUNCTION AS THE NESTED select STATEMENT ON THE PREVIOUS SLIDE, BUT WITHOUT REPETITIVE CODING. THE LOOP EXPRESSES THE PATTERN: ENTRIES ARE CONSIDERED IN ORDER OF DECREASING PRECEDENCE UNTIL A WAITING ENTRY CALL IS FOUND OR ALL ENTRIES HAVE BEEN CONSIDERED. ON EACH REPETITION OF THE LOOP, THE NULL else PART OF THE select STATEMENT IS SELECTED IF THERE IS NO ENTRY CALL WAITING ON THE SPECIFIED ENTRY FAMILY MEMBER.

  POINT OUT THAT THE for LOOP PARAMETER IS USED TO IDENTIFY THE ENTRY TO BE CONSIDERED FOR EACH EXECUTION OF THE select STATEMENT.

SOLUTION USING AN ENTRY FAMILY

```
package Buffer_Allocation_Package is

   type Buffer_Type is array (1 .. 1024) of Float;
   type Buffer_Pointer_Type is access Buffer_Type;
   type Precedence_Type is (Background, Normal, Expedited, Emergency);

   task Buffer_Allocation_Task is
      entry Obtain_Buffer (Precedence_Type)
         (Buffer_Pointer : out Buffer_Pointer_Type);
      entry Return_Buffer (Buffer_Pointer : in Buffer_Pointer_Type);
   end Buffer_Allocation_Task;

end Buffer_Allocation_Package;
```

- Obtain_Buffer IS AN ENTRY FAMILY WITH FOUR MEMBERS.

- Buffer_Allocation_Task HAS FIVE ENTRIES: Obtain_Buffer (Background), Obtain_Buffer (Normal), Obtain_Buffer (Expedited), Obtain_Buffer (Emergency), and Return_Buffer.

- REVISED LOOP:

```
while Available_Count > 0 loop
   Request_Found := False;
   for Precedence in reverse Precedence_Type loop
      select
         accept Obtain_Buffer (Precedence)
            (Buffer_Pointer : out Buffer_Pointer_Type) do
            Buffer_Pointer := Buffer_Pool (Available_Count);
         end Obtain_Buffer;
         Available_Count := Available_Count - 1;
         Request_Found := True;
         exit;
      else
         null;
      end select;
   end loop;
   exit when not Request_Found;
end loop;
```

AN ENTRY FAMILY DECLARATION GOES IN A TASK OR TASK TYPE DECLARATION, JUST LIKE AN ORDINARY ENTRY DECLARATION,

- BULLET 2: A DISCRETE SUBTYPE IS A SUBTYPE OF AN INTEGER TYPE OR AN ENUMERATION TYPE. (AS A SPECIAL CASE, A DECLARED TYPE IS CONSIDERED A SUBTYPE CONSISTING OF EVERY VALUE STIPULATED IN THE DECLARATION.)

  IN THE THIRD FORM OF THE DISCRETE RANGE, WHEN THE TWO EXPRESSIONS ARE INTEGER LITERALS, INTEGER NAMED NUMBERS, OR ATTRIBUTES WITH UNIVERSAL INTEGER VALUES, THE PREDEFINED TYPE Integer IS ASSUMED.

  DISCRETE RANGES ARE ALSO USED IN for LOOPS, SLICES, ARRAY TYPE DECLARATIONS, INDEX CONSTRAINTS, AND THE CHOICE LISTS OF CASE STATEMENTS, VARIANT PARTS, AND AGGREGATES.

- BULLET 3: EXAMPLES 1, 2, AND 3 CORRESPOND TO FORMS 1, 2, AND 3 OF BULLET 2, RESPECTIVELY. Precedence_Type IS AS DECLARED ON THE PREVIOUS SLIDE. Button_Type IS ASSUMED TO BE SOME DISCRETE TYPE FOR IDENTIFYING BUTTONS ON A CONTROL PANEL.

ENTRY FAMILY DECLARATIONS

- GENERAL FORM:

  entry [identifier] ([discrete range]) [([formal parameter list])];

- FORMS OF A [discrete range]:

  [discrete subtype name] range [expression] .. [expression]

  [discrete subtype name]

  [expression] .. [expression]

- EXAMPLES:

```
entry Obtain_Buffer (Precedence_Type range Background .. Emergency)
   (Buffer_Pointer : out Buffer_Pointer_Type);

entry Obtain_Buffer (Precedence_Type)
   (Buffer_Pointer : out Buffer_Pointer_Type);

entry Obtain_Buffer (Background .. Emergency)
   (Buffer_Pointer : out Buffer_Pointer_Type);
```

EQUIVALENT

```
entry Wait_For_Button (Button_Type);  -- FAMILY OF ENTRIES WITH NO PARAMETERS
```

INSTRUCTOR NOTES

THE SECOND EXAMPLE SHOWS THAT THE ENTRY FAMILY INDEX CAN BE COMPUTED DYNAMICALLY. (IT ALSO SHOWS THAT ACTUAL PARAMETERS CAN BE NAMED.)

SYNTACTICALLY, A CALL ON A MEMBER OF AN ENTRY FAMILY WITH NO PARAMETERS LOOKS LIKE A CALL ON AN ORDINARY ENTRY WITH ONE PARAMETER. HOWEVER, THE MEANING IS QUITE DIFFERENT.

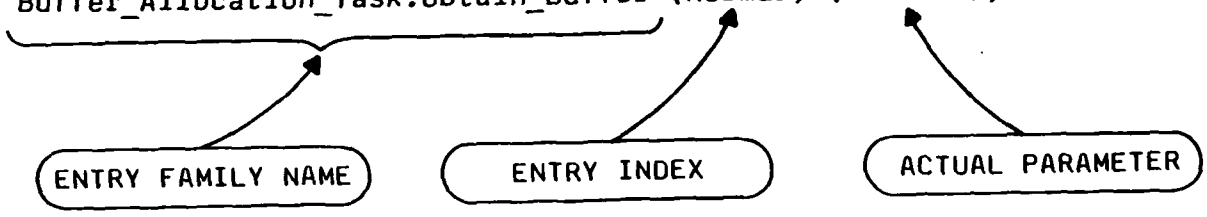
CALLING A MEMBER OF AN ENTRY FAMILY

- GENERAL FORM:

  task name . entry family identifier (expression)[(actual parameter list)];

- EXAMPLES:

```
Buffer_Allocation_Task.Obtain_Buffer (Normal) (Pointer);
```




```
Buffer_Allocation_Task.Obtain_Buffer (Precedence_Type'Succ (P))
      (Buffer_Pointer => Pointer);

Keypad_Task.Wait_For_Button (Clear_Button);
      -- CALL ON THE PARAMETERLESS ENTRY Wait_For_Button (Clear_Button)
```

INSTRUCTOR NOTES

- BULLET 1: AN accept STATEMENT OF THIS FORM IS ALLOWED ANYWHERE AN "ORDINARY" accept STATEMENT IS ALLOWED, INCLUDING IN SELECTIVE WAITS.

  AS USUAL, WE SUGGEST NEVER OMITTING THE CLOSING IDENTIFIER.

- BULLET 2: THE SECOND AND THIRD EXAMPLES ACCEPT A CALL ON A MEMBER OF AN ENTRY FAMILY WITH NO PARAMETERS. (Clear_Button IS ASSUMED TO BE AN ENUMERATION LITERAL OR CONSTANT OF TYPE Button_Type.)

  THE THIRD EXAMPLE SHOWS THAT THE ENTRY TO BE CONSIDERED MAY BE DETERMINED DYNAMICALLY.

- BULLET 3: IN A SELECTIVE WAIT, ALL ENTRY INDEXES IN THE accept STATEMENTS AT THE BEGINNINGS OF OPEN ALTERNATIVES ARE EVALUATED ONCE, AT THE BEGINNING OF THE SELECTIVE WAIT.

  ON ANY ONE EXECUTION OF A SELECTIVE WAIT OR AN accept STATEMENT, AN accept STATEMENT CAN ONLY BE WAITING FOR A CALL ON A PARTICULAR ENTRY FAMILY MEMBER, IDENTIFIED BEFORE THE WAITING BEGINS.

  HOWEVER, ON DIFFERENT EXECUTIONS OF THE SELECTIVE WAIT OR accept STATEMENT, THE accept STATEMENT MAY WAIT FOR DIFFERENT MEMBERS OF THE ENTRY FAMILY.

ACCEPTING CALLS ON ENTRY FAMILY MEMBERS

- GENERAL FORMS:

```
accept identifier ( expression )[( formal parameter list )] do
  sequence of statements
end [identifier];
accept identifier ( expression )[( formal parameter list )];
```

- EXAMPLES:

```
accept Obtain_Buffer (Precedence) (Buffer_Pointer : out Buffer_Pointer_Type) do
   Buffer_Pointer := Buffer_Pool (Available_Count);
end Obtain_Buffer;
```

JUST THE FAMILY NAME, NO INDEX

```
accept Wait_For_Button (Clear_Button);

accept Wait_For_Button (Button_Type'Succ (B)) do
   Button_Predecessor := B;
end Wait_For_Button;
```

- *MEANING:*

  THE INDEX IS EVALUATED FIRST, THEN THE accept STATEMENT WAITS FOR A CALL ON THE INDEXED ENTRY ACCORDING TO THE USUAL RULES.

INSTRUCTOR NOTES

- BULLET 2:

  THIS AVOIDS CONFUSION BETWEEN A CALL ON AN ENTRY FAMILY MEMBER WITH NO PARAMETERS AND A CALL ON A SINGLE ENTRY WITH ONE PARAMETER.

- BULLET 3:

  ONE SPECIFIED ENTRY FAMILY MEMBER IS RENAMED AS ONE PROCEDURE.

OTHER RULES

- IN A SELECTIVE WAIT, ALL ENTRY INDEX EXPRESSIONS ARE EVALUATED BEFORE WAITING BEGINS.

- ENTRY FAMILY NAMES MAY NOT BE OVERLOADED
  - WITH NAMES OF OTHER ENTRY FAMILIES FOR THE SAME TASK TYPE
  - WITH NAMES OF SINGLE ENTRIES FOR THE SAME TASK TYPE

- MEMBERS OF ENTRY FAMILIES MAY BE RENAMED AS PROCEDURES:

```
procedure Demand_Buffer (Buffer_Pointer : out Buffer_Pointer_Type)
   renames Buffer_Allocation_Task.Obtain_Buffer (Emergency);
```

INSTRUCTOR NOTES

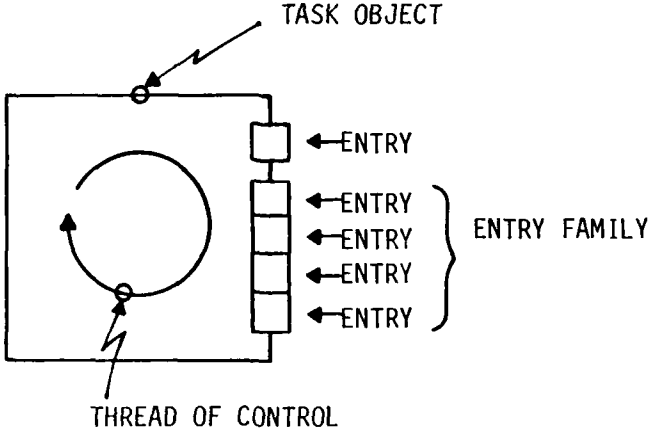
- BULLET 1:

  BY "ACCEPTED," WE REALLY MEAN "POTENTIALLY ACCEPTED."

  - ITEM 3: ARRAYS OF TASKS ARE USEFUL FOR MODELLING IDENTICAL REAL-WORLD ACTIVITIES. ENTRY FAMILIES ARE USEFUL FOR INTERACTING WITH A TASK IN ANY OF A NUMBER OF UNIFORM WAYS.

- BULLET 2:

  LEGEND:

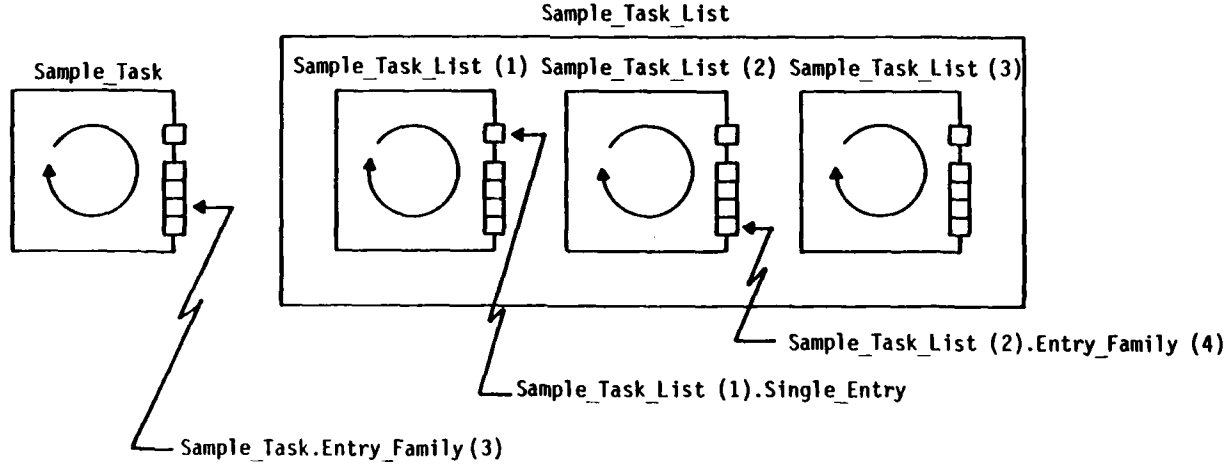


ENTRY FAMILIES VERSUS ARRAYS OF TASKS

- DO NOT CONFUSE ENTRY FAMILIES WITH ARRAYS OF TASKS.
  - ALL MEMBERS OF A TASK OBJECT'S ENTRY FAMILY ARE ACCEPTED BY THE SAME THREAD OF CONTROL.
  - IN AN ARRAY OF TASKS, CORRESPONDING ENTRIES OF DIFFERENT COMPONENTS ARE ACCEPTED BY DIFFERENT THREADS OF CONTROL.
  - ENTRY FAMILIES AND ARRAYS OF TASKS SOLVE DIFFERENT PROBLEMS.
- EXAMPLE:

```
task type Sample_Task_Type is
   entry Single_Entry;
   entry Entry_Family (1 .. 4);
end Sample_Task_Type;

Sample_Task      : Sample_Task_Type;
Sample_Task_List : array (1 .. 3) of Sample_Task_Type;
```



INSTRUCTOR NOTES

- BULLET 3:
  - ITEM 4: BITS 4 .. 7 ARE UNUSED. IT IS ASSUMED THAT BIT i OF A ONE-BYTE Bit_Sequence_Type VALUE b CORRESPONDS TO COMPONENT b(i).

EXAMPLE 1: INDEPENDENT PROCESSING OF MULTIPLEXED DATA

- PROBLEM:
  - A SURVEILLANCE SYSTEM CONTAINS FOUR CAMERAS.
  - AFTER SHOOTING A FRAME, A CAMERA BECOMES UNREADY TO SHOOT ANOTHER FRAME FOR A BRIEF BUT UNPREDICTABLE AMOUNT OF TIME.
  - THERE IS A DIFFERENT TASK CONTROLLING EACH CAMERA. EACH TASK CALLS A PROCEDURE THAT FORCES THE TASK TO WAIT UNTIL ITS CAMERA IS READY.
- EXTERNAL INTERFACE:

```
package Camera_Control_Package is
   type Camera_Number_Type is range 1 .. 4;
   procedure Wait_Until_Ready (Camera_Number : in Camera_Number_Type);
end Camera_Control_Package;
```

- ASSUMPTIONS:
  - AT REGULAR AND FREQUENT INTERVALS, A MULTIPLEXER ASSEMBLES A PACKET OF FLAGS INDICATING WHETHER EACH OF THE CAMERAS IS READY.
  - WHEN A FULL PACKET HAS BEEN ASSEMBLED, THE MULTIPLEXER ISSUES AN INTERRUPT TO VECTOR 16#35#.
  - Low_Level_IO PROVIDES A TYPE Bit_Sequence_Type FOR DATA, DECLARED AS FOLLOWS:

```
type Bit_Sequence_Type is array (0 .. 7) of Boolean;
pragma Pack (Bit_Sequence_Type);
for Bit_Sequence_Type'Size use 8;
```

  - A Receive_Control OPERATION FROM DEVICE 16#10# READS A PACKET OF FLAGS FROM ALL FOUR SENSORS INTO BITS 0 .. 3 OF A Bit_Sequence_Type VALUE.

INSTRUCTOR NOTES

INTERNALLY, Wait_Until_Ready IS IMPLEMENTED IN TERMS OF A TASK. THE TASK HAS ONE Wait ENTRY FOR EACH CAMERA, PLUS AN ENTRY FOR THE MULTIPLEXER INTERRUPT. THE Wait ENTRIES ARE PART OF AN ENTRY FAMILY. Wait_Until_Ready SIMPLY CALLS THE APPROPRIATE MEMBER OF THIS ENTRY FAMILY, USING ITS PARAMETER AS AN ENTRY INDEX.

THE PACKAGE HIDES Camera_Control_Task AND ITS ENTRIES (INCLUDING THE INTERRUPT ENTRY).

Camera_Control_Package BODY

```
with System;

package body Camera_Control_Package is

   task Camera_Control_Task is
      entry Wait (Camera_Number_Type);            -- family of ordinary entries
      entry Multiplexer_Interrupt;                -- single interrupt entry
      for Multiplexer_Interrupt use at 16#35#;
   end Camera_Control_Task;

   task body Camera_Control_Task is separate;     -- subunit on next slide

   procedure Wait_Until_Ready (Camera_Number : in Camera_Number_Type) is
   begin
      Camera_Control_Task.Wait (Camera_Number);  -- call on an entry family member
   end Wait_Until_Ready;

end Camera_Control_Package;
```

INSTRUCTOR NOTES

EACH MEMBER OF THE ENTRY FAMILY HAS ITS OWN ENTRY QUEUE, AND EACH accept ALTERNATIVE HAS ITS OWN GUARD. A FALSE GUARD FOR ONE ENTRY FAMILY MEMBER DOES NOT BLOCK RENDEZVOUS WITH ANOTHER ENTRY FAMILY MEMBER.

THE accept ALTERNATIVES SET THE BITS TO False BECAUSE A CAMERA MUST BE ASSUMED UNREADY BETWEEN THE TIME A WAIT FOR THAT CAMERA ENDS AND THE TIME THAT NEW DATA IS READ FROM THE MULTIPLEXER.

- BULLET:

  DESPITE THIS LIMITATION, THE ENTRY FAMILY SIMPLIFIED THE Wait_Until_Ready PROCEDURE BODY ON THE PREVIOUS SLIDE.

```
with Low_Level_IO;
separate (Camera_Control_Package)
task body Camera_Control_Task is
   Ready_Bits : Low_Level_IO.Bit_Sequence_Type := (0 .. 7 => False);
begin
   loop
      select
         when Ready_Bits (0) =>
            accept Wait (1);
            Ready_Bits (0) := False;
      or
         when Ready_Bits (1) =>
            accept Wait (2);
            Ready_Bits (1) := False;
      or
         when Ready_Bits (2) =>
            accept Wait (3);
            Ready_Bits (2) := False;
      or
         when Ready_Bits (3) =>
            accept Wait (4);
            Ready_Bits (3) := False;
      or
         accept Multiplexer_Interrupt;
         Low_Level_IO.Receive_Control (Device => 16#10#, Data => Ready_Bits);
      end select;
   end loop;
end Camera_Control_Task;
```

- A LIMITATION OF ENTRY FAMILIES:

  - NO WAY TO EXPRESS A PATTERN LIKE

    "WAIT FOR A CALL ON WHICHEVER OF Wait (1), Wait (2), Wait (3), Wait (4) IS CALLED FIRST, PROVIDED THAT THE CORRESPONDING BIT IN Ready_Bits IS True. THEN SET THAT BIT TO False."

  - MUST SPELL OUT FOUR VIRTUALLY IDENTICAL SELECT ALTERNATIVES.

INSTRUCTOR NOTES

THIS PACKAGE CONTROLS ALLOCATION OF RADIO FREQUENCIES TO DIFFERENT TASKS IN A COMMUNICATIONS SYSTEM. THE PACKAGE PROVIDES A PROCEDURE Reserve_Frequencies FOR RESERVING UP TO FIVE RADIO FREQUENCIES. AN ARRAY IN UNCONSTRAINED ARRAY TYPE IS FILLED WITH THE FREQUENCIES RESERVED. THE PROCEDURE USES THE LENGTH OF THE ARRAY TO DETERMINE THE NUMBER OF FREQUENCIES REQUESTED. (DUE TO THE DECLARATION OF Frequency_Count_Subtype, A Frequency_List_Type ARRAY CAN HAVE FROM 0 TO 5 COMPONENTS.)

INTERNALLY, THE PACKAGE IS IMPLEMENTED WITH A MONITOR TASK. THE MONITOR HAS SINGLE ENTRIES NAMED Request AND Release AND AN ENTRY FAMILY NAMED Reserve. A CALL ON THE Reserve_Frequencies PROCEDURE TO OBTAIN n FREQUENCIES CONSISTS OF A CALL ON Request FOLLOWED BY A CALL ON Reserve (n).

IT IS WHEN ACCEPTING A CALL ON Reserve (n) THAT Frequency_Monitor ACTUALLY ASSIGNS FREQUENCIES AND MARKS THEM AS UNAVAILABLE. DEPENDING ON WHETHER ENOUGH FREQUENCIES ARE INITIALLY AVAILABLE, A CALL ON Request WILL IMMEDIATELY CAUSE THE CALL ON Reserve (n) TO BE ACCEPTED, OR IT WILL LEAVE THAT CALL QUEUED. IN THE LATTER CASE, THE TASK CALLING Reserve_Frequencies REMAINS WAITING INSIDE THE PROCEDURE BODY, AT THE SECOND ENTRY CALL. A CALL ON Release CAUSES AS MANY QUEUED CALLS ON Reserve (n) TO BE ACCEPTED AS CAN BE, NOW THAT MORE RESOURCES ARE AVAILABLE. THE QUEUES FOR THE LARGEST FILLABLE REQUESTS ARE EXAMINED FIRST. (FIRST QUEUED CALLS ON Reserve (5), THEN QUEUED CALLS ON Reserve (4), ETC.)

DIRECT THE CLASS TO THE Reserve_Frequencies PROCEDURE BODY. MAKE SURE THEY REALIZE THAT Frequency_List'Length IS AN ACTUAL PARAMETER IN THE FIRST ENTRY CALL AND AN ENTRY INDEX IN THE SECOND.

EXAMPLE 2: RESOURCE REQUESTS OF DIFFERENT SIZES

```
package Frequency_Allocation_Package

   type Frequency_Type is (Band_1, Band_2, Band_3, Band_4, Band_5);
   subtype Frequency_Count_Subtype is Integer range 1 .. 5;
   type Frequency_List_Type is
      array (Frequency_Count_Subtype range <>) of Frequency_Type;

   procedure Reserve_Frequencies (Frequency_List : out Frequency_List_Type);
   procedure Release_Frequencies (Frequency_List : in Frequency_List_Type);

end Frequency_Allocation_Package;

package body Frequency_Allocation_Package

   task Frequency_Monitor is
      entry Request (Request_Size : in Frequency_Count_Subtype);
      entry Reserve (Frequency_Count_Subtype) (Frequency_List : out Frequency_List_Type);
      entry Release (Frequency_List : in Frequency_List_Type);
   end Frequency_Monitor;

   task body Frequency_Monitor is separate;     -- SUBUNIT ON NEXT SLIDE

   procedure Reserve_Frequencies (Frequency_List : out Frequency_List_Type) is
   begin
      Frequency_Monitor.Request (Frequency_List'Length);
      Frequency_Monitor.Reserve (Frequency_List'Length) (Frequency_List);
   end Reserve_Frequencies;

   procedure Release_Frequencies (Frequency_List : in Frequency_List_Type) is
   begin
      Frequency_Monitor.Release (Frequency_List);
   end Release_Frequencies;

end Frequency_Allocation_Package;
```

INSTRUCTOR NOTES

THE accept STATEMENT FOR Request TAKES n AS AN in PARAMETER AND CHECKS WHETHER n FREQUENCIES ARE CURRENTLY AVAILABLE. IF SO, IT IMMEDIATELY ACCEPTS A CALL ON Reserve (n); IF NOT, IT LEAVES THE CALL ON Reserve (n) QUEUED.

THE accept STATEMENT FOR Release FIRST RETURNS THE RELEASED FREQUENCIES TO THE POOL OF RELEASED FREQUENCIES. IF THIS LEAVES m FREQUENCIES AVAILABLE FOR ALLOCATION, A for LOOP THEN ACCEPTS ANY PENDING CALLS ON Reserve (m), Reserve (m-1), ETC., CONTINUING TO FILL REQUESTS IN ORDER OF DESCENDING REQUEST SIZE UNTIL THERE ARE NO MORE PENDING REQUESTS THAT CAN BE FILLED WITH THE REMAINING RESOURCES.

THE ASSIGNMENT TO Available_Count INSIDE EACH accept STATEMENT FOR THE ENTRY Reserve COULD BE MOVED TO JUST OUTSIDE THE accept STATEMENT, THUS SHORTENING THE RENDEZVOUS. THIS WAS NOT DONE BECAUSE THE ASSIGNMENT TO Frequency_List AND THE ASSIGNMENT TO Available_Count ARE CONCEPTUALLY PART OF THE SAME ABSTRACT OPERATION ON THE LIST OF AVAILABLE RESOURCES.

```
task body Frequency_Monitor is
   Available_List  : Frequency_List (1 .. 5) := (Band_1, Band_2, Band_3, Band_4, Band_5);
   Available_Count : Integer range 0 .. 5 := 5;
begin
   loop
      select
         accept Request (Request_Size : in Frequency_Count_Subtype) do
            if Request_Size <= Available_Count then
               accept Reserve (Request_Size) (Frequency_List : out Frequency_List_Type) do
                  Frequency_List :=
                     Available_List (Available_Count-Request_Size + 1 .. Available_Count);
                  Available_Count := Available_Count - Request_Size;
               end Reserve;
            end if;
         end Request;
      or
         accept Release (Frequency_List : in Frequency_List_Subtype) do
            Available_List (Available_Count + 1 .. Available_Count + Frequency_List'Length) :=
               Frequency_List;
            Available_Count := Available_Count + Frequency_List'Length;
         end Release;
         for Next_Try in reverse 1 .. Available_Count loop
            while Available_Count>= Next_Try loop
               select
                  accept Reserve (Next_Try) (Frequency_List : out Frequency_List_Type) do
                     Frequency_List :=
                        Available_List (Available_Count-Request_Size + 1 .. Available_Count);
                     Available_Count := Available_Count - Request_Size;
                  end Reserve;
               else
                  exit;
               end select;
            end loop;
         end loop;
      end select;
   end loop;
end Frequency_Monitor;
```

INSTRUCTOR NOTES

ALLOW 20 MINUTES FOR THIS SECTION.

Section 20

TASK PRIORITIES

INSTRUCTOR NOTES

SPECIFYING PRIORITIES GIVES THE RUNTIME SYSTEM A GENERAL STRATEGY TO FOLLOW.

THE NEXT SLIDE EXPLAINS HOW PRIORITIES ARE SPECIFIED. SUBSEQUENT SLIDES PRECISELY DESCRIBE THE EFFECT OF SPECIFYING A PRIORITY.

THE PURPOSE OF PRIORITIES

- A TASK TYPE CAN BE ASSIGNED A PRIORITY. THIS IS A NUMBER INDICATING THE RELATIVE URGENCY OF TASKS IN THIS TYPE, COMPARED WITH OTHER TASKS.

- PRIORITIES HELP THE RUNTIME SYSTEM ALLOCATE PROCESSORS TO TASKS WHEN THERE ARE MORE TASKS READY TO EXECUTE THAN AVAILABLE PROCESSORS.

- THIS IS THE ONLY PURPOSE OF PRIORITIES. THEY ARE NOT INTENDED FOR EXACT CONTROL OF TASK SYNCHRONIZATION.

INSTRUCTOR NOTES

- BULLET 2:

  IN THE DECLARATIVE PART OF THE MAIN PROGRAM, THE PRAGMA SPECIFIES A PRIORITY FOR THE TASK EXECUTING THE MAIN PROGRAM. THIS IS THE ONLY TASK IN AN Ada PROGRAM WITH NEITHER A TASK DECLARATION NOR A TASK TYPE DECLARATION.

- BULLET 4:

  IF NO PRIORITY IS SPECIFIED FOR A TASK, THE TASK RUNS AT AN ARBITRARY PRIORITY ASSIGNED BY THE IMPLEMENTATION.

- BULLET 5:

  EMPHASIZE ITEM 2.

THE PRIORITY PRAGMA

- FORM

  pragma *Priority* (*expression*);

- PLACEMENT:

  - IN A TASK DECLARATION OR TASK TYPE DECLARATION

```
task type Sensor_Task is
   entry Get_Reading (Reading : out Float);
   pragma Priority (5);
end Sensor_Task;
```

  - *IN THE DECLARATIVE PART OF THE MAIN PROGRAM*

- THE EXPRESSION BELONGS TO THE SUBTYPE Priority PROVIDED BY THE PACKAGE System.

  - System.Priority IS A SUBTYPE OF TYPE Integer.

  - THE LOWER AND UPPER BOUNDS OF THIS SUBTYPE DEPEND ON THE IMPLEMENTATION.

- HIGHER NUMBER = GREATER DEGREE OF URGENCY.

- THE EXPRESSION MUST BE STATIC (ABLE TO BE EVALUATED AT COMPILE-TIME).

  - TYPICALLY AN INTEGER LITERAL, A NAMED NUMBER, OR ONE OF THE ATTRIBUTES System.Priority'First, System.Priority'Last.

  - A TASK'S OR TASK TYPE'S PRIORITY CANNOT BE COMPUTED OR CHANGED WHILE A PROGRAM IS RUNNING.

INSTRUCTOR NOTES

- BULLET 1:

  THIS EXCERPT IS FROM SECTION 9.8, PARAGRAPH 4, OF THE Ada REFERENCE MANUAL. THE WORDS "COULD SENSIBLY BE EXECUTED USING THE SAME PHYSICAL PROCESSORS AND THE SAME OTHER PROCESSING RESOURCES" ACCOMMODATES SPECIAL PURPOSE PROCESSORS, SUCH AS FLOATING POINT PROCESSORS. (IF A HIGH PRIORITY TASK CAN ONLY RUN USING A FLOATING POINT PROCESSOR AND THAT PROCESSOR IS NOT AVAILABLE, A LOW PRIORITY TASK CAN CONTINUE TO RUN.) SIMILARLY, IF AN UNBLOCKED HIGH PRIORITY TASK REQUIRES MORE MEMORY THAN IS CURRENTLY AVAILABLE, A LOW PRIORITY TASK MAY CONTINUE TO RUN.

  - ITEM 1: A TASK IS ALSO INELIGIBLE FOR EXECUTION IF IT IS DECLARED IN SOME DECLARATIVE PART AND EITHER (1) THE ELABORATION OF THAT DECLARATIVE PART IS NOT COMPLETE, OR (2) THE TASK HAS FINISHED ELABORATING ITS OWN TASK BODY'S DECLARATIVE PART, BUT OTHER TASK OBJECTS DECLARED IN THE SAME PLACE HAVE NOT DONE LIKEWISE. (SEE SLIDE 17-2.)

- BULLET 2:

  "EXPIRATION OF A DELAY" REFERS TO A DELAY IN AN ORDINARY delay STATEMENT, A delay ALTERNATIVE OF A SELECTIVE WAIT, OR A TIMED ENTRY CALL.

- BULLET 3:

  - ITEM 2: WITH ONE PROCESSOR, EXPIRATION OF A DELAY IS THE ONLY REASON THAT A TASK MAY HAVE TO BE PREEMPTED ASYNCHRONOUSLY (I.E., NOT AS A DIRECT RESULT OF AN ACTION TAKEN BY THAT TASK). WITH MULTIPLE PROCESSORS, THIS CAN HAPPEN IN OTHER WAYS.

WHAT PRIORITIES DO

- OFFICIAL DEFINITION:

  "IF TWO TASKS WITH DIFFERENT PRIORITIES ARE BOTH ELIGIBLE FOR EXECUTION AND COULD SENSIBLY BE EXECUTED USING THE SAME PHYSICAL PROCESSORS AND THE SAME OTHER PROCESSING RESOURCES, THEN IT CANNOT BE THE CASE THAT THE TASK WITH THE LOWER PRIORITY IS EXECUTING WHILE THE TASK WITH THE HIGHER PRIORITY IS NOT."

  - "ELIGIBLE FOR EXECUTION" EXCLUDES TASKS THAT ARE WAITING FOR A RENDEZVOUS, THE EXPIRATION OF A DELAY, OR THE TERMINATION OF A DEPENDENT TASK.

- IMPLICATIONS FOR A ONE-PROCESSOR SYSTEM:

  - WHILE A LOWER-PRIORITY TASK IS EXECUTING, A HIGHER-PRIORITY TASK MAY BECOME ELIGIBLE FOR EXECUTION, DUE TO:

    - DECLARATION OR ALLOCATION OF THE HIGHER-PRIORITY TASK.
    - COMPLETION OF AN accept STATEMENT FOR AN ENTRY CALL MADE BY THE HIGHER-PRIORITY TASK.
    - EXPIRATION OF A DELAY.

  - THE RUNTIME SYSTEM MUST THEN SUSPEND EXECUTION OF THE LOWER-PRIORITY TASK AND RESUME EXECUTION OF THE HIGHER-PRIORITY TASK.

  - A HIGH-PRIORITY TASK MAY TERMINATE; OR IT MAY SUSPEND ITSELF TO

    - WAIT FOR A RENDEZVOUS
    - WAIT FOR A DELAY TO EXPIRE
    - WAIT FOR A DEPENDENT TASK TO TERMINATE

    THE RUNTIME SYSTEM THEN ALLOCATES THE PROCESSOR TO A LOWER-PRIORITY UNBLOCKED TASK, BASED ON THE RELATIVE PRIORITIES OF ALL UNBLOCKED TASKS.

- ADDITIONAL IMPLICATIONS FOR A MULTI-PROCESSOR SYSTEM:

  - TASKS OF DIFFERENT PRIORITIES MAY BE EXECUTING SIMULTANEOUSLY

  - A TASK OF PRIORITY 2 MAY ENTER A RENDEZVOUS THAT UNBLOCKS A TASK OF PRIORITY 3. THIS MAY FORCE A THIRD TASK, OF PRIORITY 1, TO RELINQUISH ITS PROCESSOR.

INSTRUCTOR NOTES

- BULLET 1:

  - ITEM 2: AN EXAMPLE WAS GIVEN ON SLIDE 19-4.

- BULLET 2:

  - ITEM 1: THE "OFFICIAL DEFINITION" ON THE PREVIOUS SLIDE ONLY DESCRIBES WHAT HAPPENS WHEN TASKS WITH DIFFERENT PRIORITIES ARE ALREADY "ELIGIBLE FOR EXECUTION."

  - ITEM 2: SELECTION STRATEGIES EMPHASIZING FAIRNESS INCLUDE A RANDOM DRAW AND LEAST-RECENTLY-SELECTED. A RUNTIME SYSTEM CAN BE CUSTOMIZED TO FOLLOW. THE MOST APPROPRIATE STRATEGY FOR A GIVEN APPLICATION, SINCE THE RULES OF Ada DO NOT CONSTRAIN HOW AN ALTERNATIVE IS SELECTED. MAKE SURE THE CLASS UNDERSTANDS THAT PRIORITIES MAY AFFECT THE SELECTION WITH SOME IMPLEMENTATIONS, BUT WILL NOT FOR OTHER IMPLEMENTATIONS.

WHAT PRIORITIES DON'T DO

- THEY DON'T AFFECT WHICH TASK'S CALL ON A GIVEN ENTRY WILL BE ACCEPTED FIRST.
  - WHEN SEVERAL TASKS CALL THE SAME ENTRY, THE ENTRY CALLS ARE QUEUED ON A FIRST-COME/FIRST-SERVED BASIS, REGARDLESS OF THE PRIORITIES OF THE CALLING TASKS.
  - ENTRY FAMILIES CAN BE USED TO ACCEPT CALLS IN ORDER OF URGENCY RATHER THAN THE ORDER IN WHICH CALLS ARE MADE. THERE CAN BE ONE ENTRY FAMILY MEMBER FOR EACH LEVEL OF URGENCY, EACH WITH ITS OWN QUEUE.
- THEY DON'T NECESSARILY AFFECT WHICH ALTERNATIVE OF A SELECTIVE WAIT IS CHOSEN.
  - A HIGH-PRIORITY TASK WAITING FOR ITS ENTRY CALL TO BE ACCEPTED BY A SELECTIVE WAIT DOES NOT BECOME "ELIGIBLE FOR EXECUTION" UNTIL AFTER ITS SELECT ALTERNATIVE IS CHOSEN, SO THE Priority PRAGMA DOES NOT APPLY.
  - A RUNTIME SYSTEM HAS THE OPTION OF BASING THE SELECTION ON THE PRIORITIES OF CALLING TASKS, BUT OTHER STRATEGIES, EMPHASIZING FAIRNESS, ARE SOMETIMES MORE APPROPRIATE.
- THEY DON'T NECESSARILY PROVIDE MUTUAL EXCLUSION
  - IN A MULTI-PROCESSOR SYSTEM, TASKS OF DIFFERENT PRIORITIES CAN BE ACTIVE AT THE SAME TIME.
  - PRIORITIES CAN'T BE RAISED AND LOWERED TO MANAGE CONTENTION FOR RESOURCES.

INSTRUCTOR NOTES

BULLETS 2 AND 3 RECAPITULATE INFORMATION GIVEN ON SLIDE 18-4. THE SCHEDULING OF INTERRUPT ENTRIES CAN NOW BE DESCRIBED IN TERMS OF PRIORITIES.

- BULLET 3:

  IF CALLS ARE PENDING ON AN INTERRUPT ENTRY AND AN ORDINARY ENTRY NAMED IN THE SAME SELECTIVE WAIT, AND THE SELECTIVE WAIT IS ENCOUNTERED, A RUNTIME SYSTEM IS NOT REQUIRED TO SELECT THE INTERRUPT ENTRY. HOWEVER, A REASONABLE RUNTIME SYSTEM WILL DO SO.

PRIORITIES AND RENDEZVOUS

- A RENDEZVOUS BETWEEN TWO TASKS IS EXECUTED AT THE HIGHER OF THE TWO TASKS' PRIORITIES.
  - PRIORITY OF AN accept STATEMENT VARIES, DEPENDING ON WHO'S CALLING ITS ENTRY.
- AN INTERRUPT ACTS AS A CALL ON AN INTERRUPT ENTRY BY A "HARDWARE TASK" WITH HIGHER PRIORITY THAN ANY ORDINARY TASK.
  - DIFFERENT INTERRUPTS MIGHT BE ASSOCIATED WITH DIFFERENT PRIORITY LEVELS.
- AN INTERRUPT IS ALWAYS HANDLED IMMEDIATELY, UNLESS -
  - A HIGHER-PRIORITY INTERRUPT IS ALREADY BEING HANDLED, OR
  - THE INTERRUPT HANDLER IS NOT READY (NOT WAITING AT AN accept STATEMENT).

INSTRUCTOR NOTES

- BULLET 2:

  - ITEM 2: FOR A LOWER-FREQUENCY ACTIVITY, THERE IS A GREATER PERIOD OF TIME DURING WHICH THE ACTIVITY CAN BE EXECUTED WITHOUT PREVENTING THE NEXT CYCLE OF THAT ACTIVITY FROM STARTING ON TIME. IF THE NEXT CYCLE IS DELAYED, THEN THAT CYCLE HAS A LONGER TIME DURING WHICH THE ACTIVITY CAN BE EXECUTED.

  - ITEM 5: FOR HIGHER-PRIORITY TASKS, THE DURATIONS GIVEN IN DELAY STATEMENTS MORE ACCURATELY REFLECT THE ACTUAL LENGTH OF THE DELAY. THIS IS BECAUSE A HIGHER-PRIORITY TASK OBTAINS THE PROCESSOR SOON AFTER THE EXPIRATION OF ITS DELAY MAKES IT ELIGIBLE FOR EXECUTION.

    THOUGH THE ABSOLUTE AMOUNT OF JITTER MAY BE GREATER FOR LOWER-FREQUENCY TASKS, JITTER AS A PERCENTAGE OF CYCLE TIME SHOULD BE UNIFORMLY LOW FOR ALL TASKS.

PRIORITIES AND CYCLIC PROCESSING

- FOR EACH ACTIVITY TO BE PERFORMED PERIODICALLY, THERE IS A TASK WITH A LOOP TO BE REPEATED AT A SPECIFIED FREQUENCY.
- LOOPS TO BE REPEATED MORE FREQUENTLY SHOULD BE GIVEN HIGHER PRIORITY.
  - IN THE LONG RUN, THERE MUST BE ENOUGH PROCESSING POWER TO PERFORM ALL ACTIVITIES AT THE REQUIRED FREQUENCY, BUT THERE MAY BE SHORT-TERM PERIODS IN WHICH THE PROCESSOR IS OVERLOADED.
  - LOWER-FREQUENCY ACTIVITIES HAVE A GREATER WINDOW WITHIN WHICH THEY CAN BE POSTPONED WITHOUT FALLING FAR BEHIND SCHEDULE.
  - IF BOTH A HIGH-FREQUENCY AND A LOW-FREQUENCY ACTIVITY ARE ELIGIBLE TO BEGIN EXECUTION, THE HIGH-FREQUENCY ONE WILL BE CHOSEN FIRST.
  - EXECUTION OF A LOW-FREQUENCY ACTIVITY MAY BE PREEMPTED WHEN IT IS TIME TO EXECUTE A HIGH-FREQUENCY ACTIVITY.
  - HIGHER-FREQUENCY ACTIVITIES WILL BE SUBJECT TO LESS JITTER, LOW-FREQUENCY ACTIVITIES TO MORE JITTER.
- ANALOGIES:
  - A MANAGER IS SPENDING SEVERAL WEEKS PREPARING AN ANNUAL REPORT. AT THE END OF EACH WEEK HE PUTS THE ANNUAL REPORT ASIDE MOMENTARILY TO WRITE A WEEKLY REPORT.
  - A SUBSCRIBER RECEIVES A NEWSPAPER EACH MORNING AND A NEWS MAGAZINE EACH WEEK, AND TRIES TO FINISH READING THE CURRENT ISSUE OF A PERIODICAL BEFORE THE NEXT ISSUE OF THAT PERIODICAL ARRIVES. EACH MORNING HE FINISHES NEWSPAPER BEFORE PICKING UP WHERE HE LEFT OFF READING THE MAGAZINE.

INSTRUCTOR NOTES

PART VI

IMPROVING PERFORMANCE

21. WHEN AND WHY TO TUNE

22. SHARED VARIABLE

23. MINIMIZING BLOCKING

24. MERGING TASKS

25. NON-CONCURRENT TUNING

26. WHAT'S BEST LEFT TO THE COMPILER

INSTRUCTOR NOTES

- ALLOW 45 MINUTES FOR THIS SECTION.

- THE MAIN MESSAGES ARE

  - TUNING IS OFTEN NECESSARY TO MEET REAL-TIME CONSTRAINTS.

  - TYPICALLY, PROGRAMS SPEND MOST OF THEIR TIME EXECUTING A SMALL PART OF THE PROGRAM.

  - WHICH PARTS OF A PROGRAM SHOULD BE TUNED OR WHETHER TUNING IS EVEN NECESSARY GENERALLY CAN ONLY BE DETERMINED AFTER ANALYZING THE PROGRAM (BY RUNNING PROFILES OR BY SOME OTHER METHOD).

  - PREMATURE TUNING MAKES PROGRAMS HARDER TO DEVELOP, UNDERSTAND AND MODIFY, AND OFTEN DOES NOT CONTRIBUTE TO IMPROVED PERFORMANCE.

Section 21

WHEN AND WHY TO TUNE

INSTRUCTOR NOTES

- BULLET 1 — AN ONBOARD COMPUTER SYSTEM FOR AN AIRCRAFT MUST REPORT AN INCOMING MISSILE IN A TIMELY MANNER. THE ONBOARD SYSTEM MIGHT ALSO BE SUBJECT TO WEIGHT CONSTRAINTS WHICH PREVENT ADDITIONAL MEMORY FROM BEING ADDED (BECAUSE OF THE HOUSING).

- BULLET 2

  - ITEM 1 — EMPHASIZE COMPLETE AND FUNCTIONALLY CORRECT. WE WILL SEE WHY THIS IS SO.

  - ITEM 2 — THESE STEPS NEED TO BE PERFORMED SEVERAL TIMES.

    - TO ENSURE THAT CHANGES DID IMPROVE THE SYSTEM, AND
    - TO FIND OUT IF ADDITIONAL PARTS OF THE PROGRAM NEED ATTENTION.

  - ITEM 3 — WHILE PROGRAM CLARITY IS IMPORTANT, A WEATHER "PREDICTION" PROGRAM THAT MERELY CONFIRMS YESTERDAY'S WEATHER IS UNREASONABLE.

REAL-TIME REQUIREMENTS

- EMBEDDED REAL-TIME SYSTEMS OFTEN MUST SATISFY CONSTRAINTS.
  - THROUGHPUT
  - DEADLINES
  - PERIODIC EXECUTION
  - JITTER
  - STORAGE SIZE
- TO MEET THESE CONSTRAINTS, SUCH SYSTEMS MAY NEED TO BE TUNED.
  - SYSTEM TUNING IS THE SEQUENCE OF ACTIONS APPLIED TO A COMPLETE AND FUNCTIONALLY CORRECT SYSTEM TO PRODUCE A FUNCTIONALLY IDENTICAL SYSTEM WHICH SATISFIES ITS CONSTRAINTS.
  - TUNING INCLUDES REPEATEDLY
    1. DETERMINING PARTS OF THE PROGRAM THAT AFFECT DESIRED ASPECTS OF SYSTEM PERFORMANCE.
    2. MODIFYING ONE OR MORE OF
       - SEQUENCE OF STATEMENTS EXECUTED
       - DATA STRUCTURES USED
       - ALGORITHMS USED
  - TUNING MAY SACRIFICE PROGRAM CLARITY
- THIS SECTION SUGGESTS A GENERAL STRATEGY FOR SYSTEM TUNING:
  - WHEN TO CONSIDER PERFORMANCE ISSUES
  - HOW TO DECIDE IF PROGRAM TUNING IS NECESSARY
  - HOW TO DETERMINE WHERE TO TUNE A PROGRAM
- THE REST OF COURSE WILL SUGGEST WAYS TO TUNE A SYSTEM.

INSTRUCTOR NOTES

- IT IS IMPORTANT FOR THE CLASS TO UNDERSTAND THE DISTINCTION BETWEEN THE TUNING PROCESS AND CONSIDERING PERFORMANCE DURING THE DESIGN PROCESS.

- BULLET 1 - THE BROWN CORPUS (BROWN UNIVERSITY) ANALYZED REPORTS CONTAINING OVER ONE MILLION WORDS - APPROXIMATELY 50,000 DISTINCT WORDS WERE DETECTED. THE CORPUS ALSO SHOWED THAT 256 MOST COMMON WORDS ACCOUNTED FOR OVER 55% OF THE OCCURRENCES.

  WHEN DESIGNING A SPELLING CHECKER, THIS INFORMATION WOULD BE AVAILABLE TO THE DESIGNER. A COMPETENT DESIGNER WOULD REALIZE THAT A WORD BY WORD

  SEARCH OF A DATABASE CONTAINING EVERY ENGLISH WORD WOULD NOT BE REALISTIC. USING THE BROWN CORPUS, THE DESIGNER WOULD RECOGNIZE THAT EACH WORD CHECKED SHOULD BE LOOKED UP IN THE 256 MOST COMMON WORDS FIRST.

- FOR THE INSTRUCTORS INFORMATION,

  COMPUTER PROGRAMS FOR SPELLING CORRECTION
  JAMES L. PETERSON, SPRINGER-VERLAG, NY, 1980

  DISCUSSES SUCH A SPELLING CHEKER AND PROVIDES REFERENCES FOR THE BROWN CORPUS.

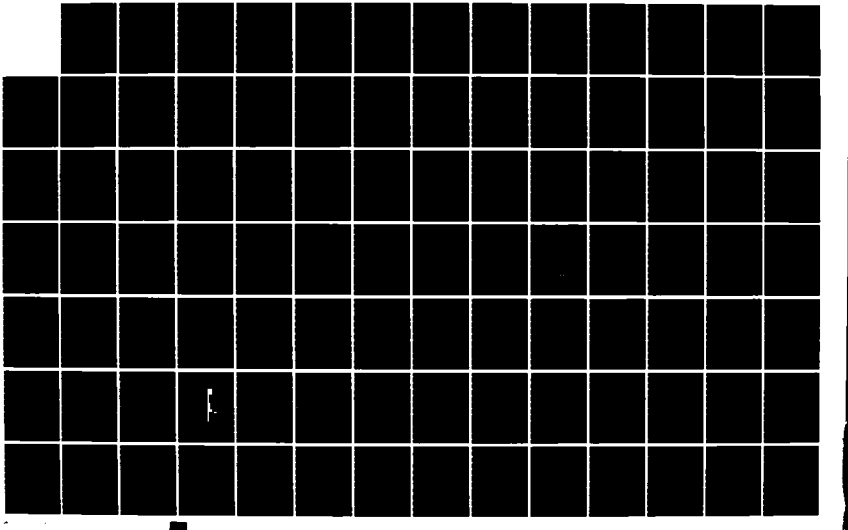
- BULLET 2 - TO CONTRAST TUNING WITH DESIGNING FOR EFFICIENCY, ANALYSIS OF THE SPELLING CHECKER MIGHT SHOW THAT SEARCHING THE 256 WORDS TAKES LONGER THAN DESIRABLE USING A BINARY SEARCH. TUNING MIGHT REQUIRE MODIFYING THE PROGRAM TO USE A DIRECTED GRAPH TO SEARCH THE WORDS.

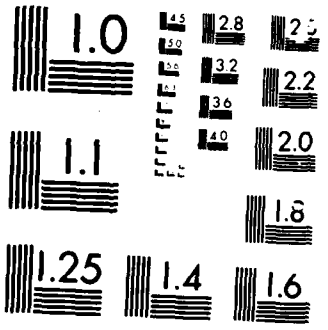
DESIGNING FOR EFFICIENCY VERSUS TUNING

- DURING THE DESIGN PROCESS, CONSTRAINTS ON A SYSTEM MAY BE CONSIDERED
  - MAY NEED TO TRADE OFF BETWEEN TIMING PERFORMANCE AND ACCURACY
  - MAY NEED TO TRADE OFF BETWEEN EXECUTION TIME AND MEMORY REQUIREMENTS
    - A SPELLING CHECKER MAY DECIDE TO KEEP THE 50,000 MOST COMMON WORDS IN ITS DATABASE. IT MAY ALSO USE A BUFFERING SCHEME TO IMPROVE EXECUTION TIME PERFORMANCE.
    - THE SPELLING CHECKER MAY KEEP THE 256 MOST FREQUENT WORDS IN MEMORY TO DECREASE EXECUTION TIME.
  - THE CONSTRAINT IN THIS EXAMPLE IS TO MAKE THE SPELLING CHECKER RESPOND IN A "TIMELY MANNER" TO THE USER.
- TUNING STARTS WITH A COMPLETE AND FUNCTIONALLY CORRECT SYSTEM
  - THERE IS AN IMPLICIT ASSUMPTION THAT ANALYSIS HAS SHOWN THAT THIS SYSTEM CAN SATISFY ITS CONSTRAINTS.
  - TUNING ACTIONS ARE USED TO MAKE THE SYSTEM MEET ITS CONSTRAINTS.

AD-A166 352 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS IN ADA L401 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 4/6

UNCLASSIFIED F/G 5/9 NL





MICROCOPY RESOLUTION TEST CHART

INSTRUCTOR NOTES

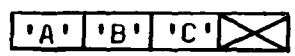
- THIS SLIDE ATTEMPTS TO LET THE CLASS KNOW THAT WE ARE NOT SAYING THAT THEY ARE TO IGNORE THEIR PRIOR EXPERIENCE OR INTUITION. IT HAS A PLACE -- IN DESIGN.

- IN THIS EXAMPLE, FELDMAN PROBABLY SUSPECTED THAT THE PROGRAM WOULD TAKE A GREAT DEAL OF TIME; HE ANALYZED THE PROGRAM AND FOUND THAT IT REQUIRED AN EXCESSIVE AMOUNT OF TIME. TRYING TO TUNE THE SIMPLE PROGRAM WOULD HAVE BEEN A WASTE OF TIME.

- FELDMAN CONSIDERED EFFICIENCY IN THE DESIGN PHASE, AND ADDRESSED IT IMMEDIATELY -- HE HAD TO!

- JUST TO MAKE SURE THE EXAMPLE IS CLEAR, THE BINARY REPRESENTATION OF THE ASCII CHARACTERS 'A', 'B' AND 'C' is

  'A' - 0100 0001 - 2 ones
  'B' - 0100 0010 - 2 ones
  'C' - 0100 0011 - 3 ones

  SO IF THE CODE WORD IS 

  WE HAVE

```
SUM :=   Count ('A')
       + Count ('B')
       + Count ('C');
```

  WHICH IS 2 + 2 + 3 = 7.

- IN THE REMAINDER OF THIS SECTION WE ARE CONCERNED WITH TUNING, NOT DESIGNING FOR EFFICIENCY

DESIGNING FOR EFFICIENCY: AN EXAMPLE

- IN HIS BOOK WRITING EFFICIENT PROGRAMS (SEE BIBLIOGRAPHY) JON BENTLEY RELATES THE FOLLOWING STORY

- S. FELDMAN (OF BELL LABS) WANTED TO WRITE A PROGRAM THAT WOULD

  - CALCULATE THE NUMBER OF 1'S IN A LONG SERIES OF 23-BIT BINARY CODEWORDS.
  - EACH CODEWORD OCCURS IN A 32-BIT WORD, FOLLOWED BY 9 ZEROES.
  - ANALYSIS SHOWED THAT A SIMPLE PROGRAM WOULD REQUIRE MORE THAN ONE WEEK OF COMPUTER TIME.
  - WRITING THE SIMPLE PROGRAM AND TRYING TO TUNE IT WOULD HAVE BEEN USELESS.

- INSTEAD FELDMAN TOOK THE FOLLOWING APPROACH

```
type Code_Word_Type is array (1 .. 4) of Character; -- FOUR BITES PER WORD
pragma Pack (Code_Word_Type);
Code_Word:  Code_Word_Type;
   ...
Ones_Count : constant array (Character) of Integer range 0 .. 8 :=
                  (0, 1, 1, 2, 1, 2, ...);
  -- Ones_Count (Ch) is a count of the number of 1's in the representation
  -- of character ch
   ...

Sum := Ones_Count (Code_Word (1))
       + Ones_Count (Code_Word (2))
       + Ones_Count (Code_Word (3));
```

- THE RESULTING PROGRAM (WITH SOME ADDITIONAL MODIFICATIONS) TOOK LESS THAN 2 HOURS.

INSTRUCTOR NOTES

- TUNING CAN MEAN USING A BETTER ALGORITHM (BINARY SEARCH RATHER THAN LINEAR SEARCH) OR IT CAN MEAN USING EFFICIENT CODING TECHNIQUES. IN EITHER CASE, WE NEED TO BE ABLE TO UNDERSTAND THE PROGRAM BEFORE WE CAN TUNE IT. WE ALSO NEED TO UNDERSTAND THE DESIGN BEFORE WE CAN PREPARE AN UNDERSTANDABLE AND CORRECT IMPLEMENTATION.

- MUST START WITH A CORRECT IMPLEMENTATION. IT MAKES NO SENSE TO EXPEND EFFORT TO MAKE A SYSTEM FAIL FASTER.

- ANYTIME YOU START TINKERING WITH A PROGRAM YOU OPEN UP THE POSSIBILITIES OF INTRODUCING ERRORS.

- FOR BULLET 4, TELL THE CLASS WE WILL DISCUSS HOW TO FIND THESE BOTTLENECKS.

STEPS TO TAKE BEFORE TUNING

- START WITH A CORRECT AND EASILY UNDERSTOOD DESIGN.

- PRODUCE A SYSTEM THAT IS A CORRECT AND EASILY UNDERSTOOD IMPLEMENTATION OF THE DESIGN.

  - IN ORDER TO TUNE A SYSTEM, IT MUST BE UNDERSTOOD.
  - TUNING OFTEN MAKES A SYSTEM MORE DIFFICULT TO UNDERSTAND, HENCE MAINTAIN.
  - TUNING A SYSTEM OPENS UP THE POSSIBILITY OF INTRODUCING ERRORS.

- DETERMINE IF PERFORMANCE IS AN ISSUE.

  "THERE IS NO SENSE IN MAKING A PROGRAM RUN FASTER IF THE RESULT IS MERELY TO INCREASE THE PROPORTION OF UNUSED CPU CYCLES."
  M.A. JACKSON, PRINCIPLES OF PROGRAM DESIGN.

- FIND THE PERFORMANCE BOTTLENECKS AND REMOVE THEM.

- LEAVE THE REST OF THE SYSTEM ALONE.

INSTRUCTOR NOTES

- BULLET 1

  WE GIVE AN EXAMPLE OF PREMATURE OPTIMIZATION PROBLEMS SHORTLY.

  - THE QUOTE IS FROM KNUTH'S ARTICLE Structured Programming with go to Statements. THE STUDY IS DESCRIBED IN An Empirical Study of FORTRAN Programs. (SEE BIBLIOGRAPHY).

- BULLET 2

  - ITEM 1 IS OBTAINED FROM KNUTH'S FORTRAN PROGRAM STUDY.

  - ITEM 2 IMPROVING THE PERFORMANCE OF THE 3% AFFECTS MORE THAN 50% OF THE EXECUTION TIME.

HOW MUCH OF A PROGRAM SHOULD BE TUNED

- DONALD KNUTH PERFORMED STUDIES ON FORTRAN PROGRAMS. HE CONCLUDED

  "...MOST OF THE RUNNING TIME IN NON-IO-BOUND PROGRAMS IS CONCENTRATED IN ABOUT 3% OF THE SOURCE TEXT... WE SHOULD FORGET ABOUT SMALL EFFICIENCIES, SAY ABOUT 97% OF THE TIME: PREMATURE OPTIMIZATION IS THE ROOT OF ALL EVIL."

- CONCENTRATE TUNING ON THE 3%.

  - ACCOUNTS FOR OVER 50% OF PROGRAM EXECUTION.
  - TUNING HERE PROVIDES THE BEST RETURN.
    - SPEEDING UP A CRITICAL LOOP BY 10% MAY SPEED UP ENTIRE PROGRAM BY NEARLY 10%.
    - SPEEDING UP OTHER CODE BY 50% MAY HAVE NO MEASURABLE EFFECT.

- NEED TO FIND THE PORTIONS OF THE PROGRAM THAT MAKE UP THE 3%.

INSTRUCTOR NOTES

- THIS IS THE FIRST OF TWO SLIDES GIVING AN EXAMPLE OF HOW PREMATURE TUNING CAN RESULT IN WASTED EFFORT.

- EMPHASIZE THAT WE ARE DISCUSSING PREMATURE TUNING, NOT INCORRECT OR UNDISCIPLINED TUNING.

A PENALTY OF PREMATURE TUNING

- BENTLEY RELATES THE FOLLOWING STORY:
- A FORTRAN COMPILER WAS BEING ENHANCED IN EARLY 1960's SUBJECT TO THE CONSTRAINT THAT USERS WOULD NOT NOTICE AN INCREASE IN COMPILATION TIME.
- A PARTICULAR SUBPROGRAM WITHIN THE COMPILER WAS RARELY USED. THE PROGRAMMER DOING THE ENHANCEMENTS ESTIMATED THAT
  - 1% OF THE COMPILATIONS USED THE ROUTINE.
  - ROUTINE USED AT MOST ONCE PER COMPILATION.
- SINCE THE SUBPROGRAM WAS VERY SLOW IT WAS TUNED.
  - TUNING TOOK 1 WEEK
  - SQUEEZED "...EVERY LAST UNNEEDED CYCLE OUT OF THE" SUBPROGRAM.
- MODIFIED COMPILER RAN FAST ENOUGH.

INSTRUCTOR NOTES

- THIS IS A FAIRLY COMMON SITUATION. PROGRAMMERS ALWAYS "THINK" THEY KNOW WHERE TUNING SHOULD OCCUR. PERFORMANCE ANALYSIS TECHNIQUES TELL US WHERE TUNING SHOULD OCCUR.

MUCH ADO ABOUT NOTHING

- AFTER 2 YEARS EXTENSIVE USE
  - COMPILER REPORTED AN INTERNAL ERROR DURING COMPILATION.
  - THE ERROR WAS IN THE PROLOGUE (STARTING CODE) OF THE "CRITICAL" SUBPROGRAM.
  - ERROR EXISTED DURING ENTIRE LIFE OF THE SUBPROGRAM.
- CONCLUSIONS REACHED WERE
  - SUBPROGRAM WAS NEVER CALLED IN THE MORE THAN 100,000 COMPILATIONS EXECUTED IN THE LIFE OF THE "ENHANCED" COMPILER.
  - THE WEEK'S EFFORT AT TUNING THE SUBPROGRAM WAS MUCH ADO ABOUT NOTHING.
- IN GENERAL, PROGRAMMERS HAVE A GOOD TRACK RECORD AT BEING VERY BAD AT GUESSING WHICH PARTS OF A PROGRAM NEED TO BE TUNED.
  - KNUTH HAS FOUND THAT

    "IT IS OFTEN A MISTAKE TO MAKE A PRIORI JUDGMENTS ABOUT WHAT PARTS OF A PROGRAM ARE REALLY CRITICAL, SINCE THE UNIVERSAL EXPERIENCE OF PROGRAMMERS WHO HAVE BEEN USING MEASUREMENT TOOLS HAS BEEN THAT THEIR INTUITIVE GUESSES FAIL."
- PERFORMANCE ANALYSIS TECHNIQUES SHOULD BE USED TO FIND WHERE PROGRAM TUNING IS NEEDED.

INSTRUCTOR NOTES

- THIS SLIDE STATES THE PROBLEMS WITH PREMATURE OPTIMIZATION IN GENERAL.

- BULLET 3

  - ITEM 2 AN EXAMPLE IS EXPLOITING THE BIT REPRESENTATION OF THE WORD PROVIDED BY A PARTICULAR SENSOR. ALGORITHM MUST CHANGE WHEN A NEW SENSOR IS INSTALLED.

PROBLEMS WITH PREMATURE TUNING

- MISDIRECTED EFFORT
  - THE FORTRAN COMPILER EXAMPLE SHOWS THAT EFFORT MIGHT BE EXPENDED NEEDLESSLY.
- LESS RELIABLE
  - MANY TIMES TUNING INVOLVES COMPLEX OR SUBTLE LOGIC.
  - COMPLEX CODE IS MORE DIFFICULT TO WRITE AND MORE ERROR-PRONE.
- LESS MAINTAINABLE
  - PROGRAM MORE DIFFICULT TO UNDERSTAND.
  - PROGRAM TAKES ADVANTAGE OF PROPERTIES OF THE PROBLEM, INCREASING THE SENSITIVITY OF THE CODE TO CHANGES IN THE PROBLEM.
  - GREATER CHANCE OF INTRODUCING ERRORS DUE TO COMPLEX/HIDDEN INTERACTIONS, OR DEPENDENCIES ON COMPLICATED ASSUMPTIONS.
- INCREASED INTERACTION CAN INCREASE RECOMPILATION DEPENDENCIES.

INSTRUCTOR NOTES

- PROFILING (WHICH IS THE ONLY METHOD WE WILL LOOK AT IN DETAIL) CAN AFFECT OVERALL EXECUTION TIME SINCE CODE IS INSERTED IN THE PROGRAM. THIS IS ESPECIALLY TRUE FOR FREQUENCY COUNTING. THE STATISTICAL PROFILE IS LESS ACCURATE, BUT IT CAN BE MORE REALISTIC IN THAT IT SHOWS HOW MUCH TIME THE PROGRAM SPENT IN SYSTEM ROUTINES. IF FIXED INTERVALS ARE USED FOR THE STATISTICAL PROFILE, THEN SEVERAL RUNS WITH DIFFERENT INTERVALS SHOULD BE OBTAINED. THIS AVOIDS THE PROBLEM OF THE SAMPLING GETTING IN SYNC WITH A LOOP.

- A PROFILE PROGRAM CREATES A COPY OF THE SOURCE AND ADDS STATEMENTS TO DO THE PROFILE. A COMPILER ADDS OBJECT CODE TO DO THE PROFILE.

- BULLET 3

  - ITEM 1

    - SUBITEM 1 THIS KIND OF ERROR MIGHT BE THE RESULT OF AN INCORRECTLY FORMULATED CONDITIONAL EXPRESSION IN AN IF-STATEMENT, WHILE-STATEMENT, ETC.

    - SUBITEM 2 DURING CODING, THE PROGRAMMER MAY HAVE FAILED TO REALIZE THAT A PARTICULAR CASE CANNOT HAPPEN. FOR EXAMPLE, TESTING IF A VARIABLE IS NEGATIVE AFTER IT HAS BEEN SQUARED.

    - SUBITEM 3 THIS MIGHT REQUIRE ADDITIONAL RUNS (POSSIBLY WITH ADDITIONAL TEST DATA). THIS DOESN'T NEED TO BE A POSITIVE PATH. IT COULD BE AN ERROR PATH. ERROR PATHS SHOULD BE CONSIDERED WHEN TUNING. A NUCLEAR POWER PLANT CONTROL PROGRAM SHOULD NOT "SLOW DOWN" IF AN ERROR OCCURS.

  - ITEM 2 SEE THE TABLE LOOK UP SLIDE #10.

- BULLET 4 - THE NEXT SLIDE DESCRIBES WHY.

DETERMINING WHERE TO TUNE: PROFILES

- ONE WAY TO DETERMINE THE PARTS OF A PROGRAM TO TUNE IS BY USING PROFILES.
  - FREQUENCY PROFILE
    - COUNTERS ARE INSERTED IN THE PROGRAM.
    - SHOWS HOW OFTEN EACH STATEMENT IS EXECUTED.
    - SHOWS HOW MUCH TIME SPENT IN EACH SUBPROGRAM.
  - STATISTICAL PROFILE
    - PROGRAM IS FREQUENTLY INTERRUPTED WITH LOCATION IN PROGRAM NOTED.
    - SHOWS HOW OFTEN THE PROGRAM WAS IN A CERTAIN RANGE OF STATEMENTS (INSTRUCTIONS).
- PROFILES MAY BE PROVIDED BY
  - COMPILER
  - PROFILE PROGRAM
  - THE ALS Ada COMPILER PROVIDES FOR BOTH FREQUENCY AND STATISTICAL PROFILES.
- SIDE BENEFITS
  - SHOWS PARTS OF PROGRAM THAT ARE NEVER EXECUTED, CALLING ATTENTION TO:
    - INCORRECTLY FORMULATED CONDITIONS
    - UNREACHABLE CODE
    - INCOMPLETE TESTING (E.G. ERROR PATHS).
  - SHOWS POSSIBLE RELATIONSHIPS BETWEEN PARTS OF PROGRAM.
- INITIAL PROFILES SHOULD BE OBTAINED BEFORE TUNING STARTS.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS THE DRAMATIC RESULTS THAT CAN BE OBTAINED BY USING A PROFILE. THIS IS NOT ATYPICAL. OF COURSE ADDITIONAL TUNING WILL NOT BE AS "EFFECTIVE" BUT IT WILL STILL IMPROVE PERFORMANCE.

- BULLETS 4 AND 5 — EMPHASIZE 20 LINES OF CODE WITH A FEW HOURS WORK.

- BULLET 6 — TELL THE CLASS THAT OBTAINING ANOTHER PROFILE NOT ONLY VERIFIES THAT TUNING CHANGES HAVE THE DESIRED EFFECT, BUT IT CAN ALSO POINT OUT ADDITIONAL PARTS OF THE PROGRAM THAT CAN BE TUNED.

PROFILE OF A SUCCESS STORY

BENTLEY RELATES THE FOLLOWING STORY:

- C. VAN WYK (OF BELL LABS) PROFILED AN INTERPRETER HE HAD DEVELOPED.
  - RAN 10 TEST CASES THAT EXECUTED EVERY PROGRAM PATH.
- THE PROFILE SHOWED
  - 70% OF THE EXECUTION TIME WAS SPENT IN THE SYSTEM MEMORY ALLOCATION SUBPROGRAM.
  - LOOKING AT THIS ROUTINE, VAN WYK FOUND
    - MOST OF THE TIME SPENT ALLOCATING ONE PARTICULAR KIND OF RECORD -- 68,000 TIMES.
    - NEXT MOST POPULAR RECORD ALLOCATED 2000 TIMES.
- VAN WYK ADDED ABOUT 20 LINES OF CODE TO KEEP FREE RECORDS OF THIS TYPE ON A QUEUE THAT THE PROGRAM MANAGED ITSELF.
  - THE MODIFIED PROGRAM'S RUN TIME DECREASED TO 45%.
  - MEMORY ALLOCATION DECREASED TO 30% OF PROGRAM EXECUTION.
- ALL OF THIS WAS ACCOMPLISHED IN A FEW HOURS ONE AFTERNOON.
- SIDE BENEFIT - OBTAINING PROFILE OF IMPROVED VERSION
  - POINTED OUT PLACES FOR ADDITIONAL TUNING.
  - THESE WERE DETECTED BECAUSE PROFILE WAS NO LONGER OVERWHELMED BY MEMORY ALLOCATION.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS WHY YOU CANNOT ACCEPT PROFILE RESULTS BLINDLY.

USE COMMON SENSE IN USING PROFILES

- WHEN YOU INTERPRET RESULTS FROM PROFILES
  - LOOK AT THE CONTEXT IN WHICH THE TIME CONSUMING PORTIONS APPEAR.
  - IF A GREAT DEAL OF TIME IS SPENT IN A SMALL SUBPROGRAM THE PROBLEM MIGHT BE THAT THE SUBPROGRAM IS CALLED TOO OFTEN.
- SUPPOSE A TABLE LOOK UP PACKAGE PROFILE SHOWS THAT 90% OF THE TIME IS SPENT IN A VARIABLE LENGTH STRING COMPARE SUBPROGRAM.
- TWO APPROACHES CAN BE USED TO ANALYZE THE RESULTS.
  - NAIVE APPROACH
    - SPEED UP THE COMPARE SUBPROGRAM
    - MAYBE GET FACTOR OF TWO SPEEDUP
  - MORE SOPHISTICATED APPROACH
    - STUDY HOW THE COMPARE SUBPROGRAM IS USED.
    - MIGHT FIND TABLE LOOK UP PACKAGE USING A LINEAR SEARCH.
    - REPLACING THE LINEAR SEARCHING WITH HASHING MIGHT GET A FACTOR TEN SPEEDUP.

INSTRUCTOR NOTES

- BULLET 1 - DEPENDS ON RUNTIME SYSTEM
  - THE FIRST PROBLEM CAN BE SOLVED IF THE RUNTIME SYSTEM ASSOCIATES A UNIQUE TASK IDENTIFIER TO EACH SYSTEM AND IF THE PROFILER HAS ACCESS TO THESE TASK ID'S (IN WHICH CASE IT CAN IDENTIFY THEM).
  - THE SECOND PROBLEM CAN BE SOLVED IF THE PROFILER CAN OBTAIN THIS INFORMATION FROM THE RUNTIME SYSTEM. IN THIS CASE, A STATISTICAL PROFILE MIGHT BE NEEDED.

- BULLET 2 - A STATISTICAL PROFILE COULD BE USED

- BULLET 3
  - IF THE PROGRAM IS FOR AN EMBEDDED SYSTEM IT WILL PROBABLY NOT HAVE THE REQUIRED FEATURES
  - PROFILING COULD INCREASE THE EXECUTION TIME BETWEEN TIME CRITICAL POINTS, THEREBY CAUSING TIMING CONSTRAINTS TO BE VIOLATED
  - TIMING MODELS ARE NOT DISCUSSED IN THIS COURSE. ESSENTIALLY A TIMING MODEL IS A MODEL OF THE SOFTWARE, AND IS USED TO DETERMINE PROGRAM EXECUTION TIME. TIMING MODELS ARE DISCUSSED IN MORE DETAIL IN "Ada REAL TIME STUDIES" 1986 CHAPTER 8.

PROBLEMS WITH PROFILES

- IN A MULTI-TASK PROGRAM
  - WHICH THREAD OF CONTROL IS REPORTED AS THE CURRENT PROGRAM POSITION?
  - IS THE TASK WAITING FOR A RENDEZVOUS OR WAITING FOR A HIGHER PRIORITY TASK TO GIVE UP THE PROCESSOR?

- HOW CAN WE DISTINGUISH BETWEEN PROGRAM AND RUNTIME SYSTEM EXECUTION SUCH AS GARBAGE COLLECTION?

- WHAT IF WE CANNOT OBTAIN PROFILES IN THE TARGET ENVIRONMENT?
  - TARGET ENVIRONMENT MIGHT NOT CONTAIN ADEQUATE FACILITIES:

    MEMORY, I/O, PROCESSING POWER
  - PROFILING MAY RESULT IN THE PROGRAM NOT BEHAVING PROPERLY, E.G., NOT MEETING TIMING CONSTRAINTS
  - STILL BENEFICIAL TO USE PROFILES IN THE HOST ENVIRONMENT, ESPECIALLY WHEN SUPPLEMENTED BY TIMING MODELS DEVELOPED TO DETERMINE IF CONSTRAINTS ARE MET

INSTRUCTOR NOTES

- BENTLEY'S COLUMN APPEARS MONTHLY IN THE COMMUNICATIONS OF THE ACM (CACM).
- SEVERAL TIMES THE SLIDES HAVE MENTIONED THAT TUNING CAN INCREASE COMPLEXITY. THIS SLIDE AND THE NEXT HAMMER THAT POINT IN!
- BINARY SEARCHES WERE COVERED IN L305, SO DO NOT DISCUSS THEM HERE. JUST POINT OUT HOW SIMPLE AND STRAIGHTFORWARD THE EXAMPLE IS.
- THE POINT THAT THIS SLIDE AND THE NEXT ARE TRYING TO MAKE IS THAT YOU NEED TO START OUT WITH AN UNDERSTANDABLE AND CORRECT PROGRAM, AND THEN TUNE IT IF NECESSARY. THE EXAMPLE ON THE NEXT PAGE IS MUCH MORE DIFFICULT TO UNDERSTAND, YOU WOULD NOT WANT TO USE IT UNLESS NECESSARY.
- FOR INSTRUCTOR ONLY
  - AN EXPLICIT VALUE OF 1000 IS SHOWN FOR AN ARRAY INDEX SINCE IT IS NEEDED IN THE NEXT SLIDE.
  - BINARY SEARCH IS O ($n \log_2 n$).

INCREASING COMPLEXITY THROUGH TUNING

- IN HIS PROGRAMMING PEARLS COLUMN, BENTLEY STARTS WITH A BINARY SEARCH OF THE FORM:

```
Table : array (1 .. 1000) of Integer;

      ...

   Lower_Index := Table'First;
   Upper_Index := Table'Last;

   loop
       if Lower_Index > Upper_Index then
          return 0;
       else
          Mid_Point := (Lower_Index + Upper_Index)/2;
          if Table (Mid_Point) < Search_Item then
              Lower_Index := Mid_Point + 1;
          elsif Table (Mid_Point) = Search_Item then
              return Mid_Point;
          else -- Table (Mid_Point) > Search_Item
              Upper_Index := Mid_Point - 1;
          end if;
       end if;
   end loop;
```

- A BINARY SEARCH IS FAST, SO IT USUALLY DOES NOT NEED TUNING.

- IN THIS CASE, THE SEARCH IS IN A CRITICAL PATH, SO THIS TIME IT IS TUNED

  - TUNING MAKES PROGRAM 2-3 TIMES FASTER.
  - CLARITY IS SACRIFICED FOR EFFICIENCY.

INSTRUCTOR NOTES

- JUST POINT OUT THAT IT WOULD BE DIFFICULT TO COME UP WITH THIS RIGHT OFF, AND THAT IF ATTEMPTED IT WOULD PROBABLY BE UNRELIABLE.

- THIS IS AN EXAMPLE OF LOOP UNFOLDING. THE LOOP IS REMOVED. THIS IS EXPLAINED IN BENTLEY'S BOOK.

- DO NOT TRY TO EXPLAIN THIS TO THE CLASS.

- FOR THE INSTRUCTOR ONLY THIS SEARCH WORKS BY BISECTING THE ARRAY. Lower_Index + 1 MARKS THE FIRST ELEMENT OF THE INTERVAL WITH Lower_Index + n THE LAST ELEMENT IN THE LEFT HAND PORTION OF THE INTERVAL (SIMILAR TO THE MIDPOINT OF THE PREVIOUS SLIDE).

```
Lower_Index := 1;
if Table (512) < Search_Item then
   Lower_Index := 1000 + 1 - 512;
end if;
if Table (Lower_Index + 256) < Search_Item then
   Lower_Index := Lower_Index + 256;
end if;
if Table (Lower_Index + 128) < Search_Item then
   Lower_Index := Lower_Index + 128;
end if;
if Table (Lower_Index + 64) < Search_Item then
   Lower_Index := Lower_Index + 64;
end if;
if Table (Lower_Index + 32) < Search_Item then
   Lower_Index := Lower_Index + 32;
end if;
if Table (Lower_Index + 16) < Search_Item then
   Lower_Index := Lower_Index + 16;
end if;
if Table (Lower_Index + 8) < Search_Item then
   Lower_Index := Lower_Index + 8;
end if;
if Table (Lower_Index + 4) < Search_Item then
   Lower_Index := Lower_Index + 4;
end if;
if Table (Lower_Index + 2) < Search_Item then
   Lower_Index := Lower_Index + 2;
end if;
if Table (Lower_Index + 1) < Search_Item then
   Lower_Index := Lower_Index + 1;
end if;
if Lower_Index <= 1000 and then Table (Lower_Index) = Search_Item then
   return Lower_Index;
else
   return 0;
end if;
```

- STARTED WITH AN UNDERSTANDABLE AND CORRECT BINARY SEARCH.
  - NEED FOR TUNING ESTABLISHED.
  - DID NOT TRY WRITING EFFICIENT VERSION FIRST.
  - ALGORITHM TRANSFORMED TO MORE EFFICIENT VERSION.

INSTRUCTOR NOTES

KEEP RECORD OF CHANGES

- KEEP RECORD OF CHANGES MADE WHEN TUNING.
  - RECORD AS COMMENTS IN THE PROGRAM.
  - GIVES A COMPLETE HISTORY OF THE CHANGES.
- CAN BE USED WHEN MODIFYING THE TUNED CODE.
  - IN THE BINARY SEARCH EXAMPLE, IF THE TABLE SIZE IS INCREASED TO 5000.
    - PROGRAMMER WILL PROBABLY FEEL MORE COMFORTABLE GOING THROUGH THE SEQUENCE OF CHANGES, STARTING WITH A TABLE SIZE OF 5000.
  - IN GENERAL, THE PROGRAMMER WILL UNDERSTAND THE EFFECT OF THE CHANGES.
    - POSSIBLY FIND INCOMPATIBILITIES WITH PLANNED MODIFICATIONS AND PREVIOUS TUNING.
    - DISCOVER NEW TUNING POSSIBILITIES.

INSTRUCTOR NOTES

JACKSON'S RULES FOR TUNING

- "RULE 1 - DON'T DO IT."
  - UNLESS IT CAN BE JUSTIFIED.
  - DON'T INCREASE UNUSED CYCLES.
- "RULE 2 - (FOR EXPERTS ONLY). DON'T DO IT YET."
  - START FROM A CLEAR, UNOPTIMIZED SOLUTION.
  - UNDERSTAND WHAT YOU ARE GOING TO CHANGE.

INSTRUCTOR NOTES

- EFFICIENCY IS IMPORTANT, BUT IT'S JUST ONE OF THE PROGRAMMING CONSIDERATIONS THAT RESULT IN GOOD PROGRAMS. STRUCTURE IS IMPORTANT, BUT AGAIN IT'S JUST ONE OF THE CONSIDERATIONS. THE SUCCESSFUL PROGRAMMER MUST MAINTAIN THE PROPER PERSPECTIVE ON BOTH CONSIDERATIONS.

- THIS IS A GOOD PLACE TO URGE THE CLASS TO READ BENTLEY'S BOOK AND TO SUGGEST THAT HIS PROGRAMMING PEARLS COLUMN SHOULD BE CONSIDERED MUST READING.

KEEPING THE PROPER PERSPECTIVE

IN PROGRAMMING PEARLS, BENTLEY WRITES

> "SOME PROGRAMMERS PAY TOO MUCH ATTENTION TO EFFICIENCY; BY WORRYING TOO SOON ABOUT LITTLE 'OPTIMIZATIONS' THEY CREATE RUTHLESSLY CLEVER PROGRAMS THAT ARE INSIDIOUSLY DIFFICULT TO MAINTAIN. OTHERS PAY TOO LITTLE ATTENTION; THEY END UP WITH BEAUTIFULLY STRUCTURED PROGRAMS THAT ARE UTTERLY INEFFICIENT AND THEREFORE USELESS. ONE HAS TO KEEP PERSPECTIVE ON EFFICIENCY: IT IS JUST ONE OF MANY PROBLEMS, BUT IT IS SOMETIMES VERY IMPORTANT."

INSTRUCTOR NOTES

- ALLOW 30 MINUTES FOR THIS SECTION.
- THIS SECTION SHOWS HOW SHARED VARIABLES CAN BE USED FOR TUNING.
- RECOMMEND THAT STUDENTS READ EXERCISE 6.1 IN THE REAL-TIME ADA WORKBOOK.

Section 22

SHARED VARIABLES

INSTRUCTOR NOTES

- BULLETS 1-3 SUMMARIZE SOME OF THE CONCLUSIONS OF SECTION 21. WE REFER TO THE PARTS OF THE PROGRAM THAT MIGHT NEED TUNING AS "HOT SPOTS."

- BULLET 4 IN ATTEMPTING TO IMPROVE EFFICIENCY, IT IS EASY TO FORGET ABOUT SAFETY. IF THE ASSUMPTIONS THAT LEAD TO TUNING CHANGES ARE ONLY VALID 95% OF THE TIME, THEN EFFICIENCY HAS BEEN GAINED UNDER FALSE PRETENSE.

  REMEMBER TUNING TAKES A CORRECT, WORKING PROGRAM AND TRANSFORMS IT INTO AN EQUIVALENT PROGRAM THAT IS MORE EFFICIENT.

- BULLET 5 WHEN EFFICIENCY REQUIRES IT, TASKS CAN SOMETIMES BENEFIT BY USING SHARED VARIABLES. EMPHASIZE THAT THIS IS BEING SUGGESTED AS A POSSIBLE TUNING TECHNIQUE, NOT A GENERAL PROGRAMMING PRACTICE.

A REVIEW OF TUNING

- A PROGRAM THAT IS
  - CORRECT
  - UNDERSTANDABLE

  CAN BE TUNED IF PERFORMANCE "HOT SPOTS" EXIST.

- PROFILES SHOULD BE USED TO FIND "HOT SPOTS."

- PREMATURE TUNING CAN LEAD TO
  - MISDIRECTED EFFORT
  - LESS RELIABLE PROGRAMS
  - LESS MAINTAINABLE PROGRAMS
  - INCREASED COMPILATION DEPENDENCIES.

- TUNING CAN INTRODUCE ERRORS IF CARE IS NOT TAKEN.
  - DO NOT TRADE EFFICIENCY FOR SAFETY.
  - MAKE SURE ASSUMPTIONS BEING MADE ARE VALID.

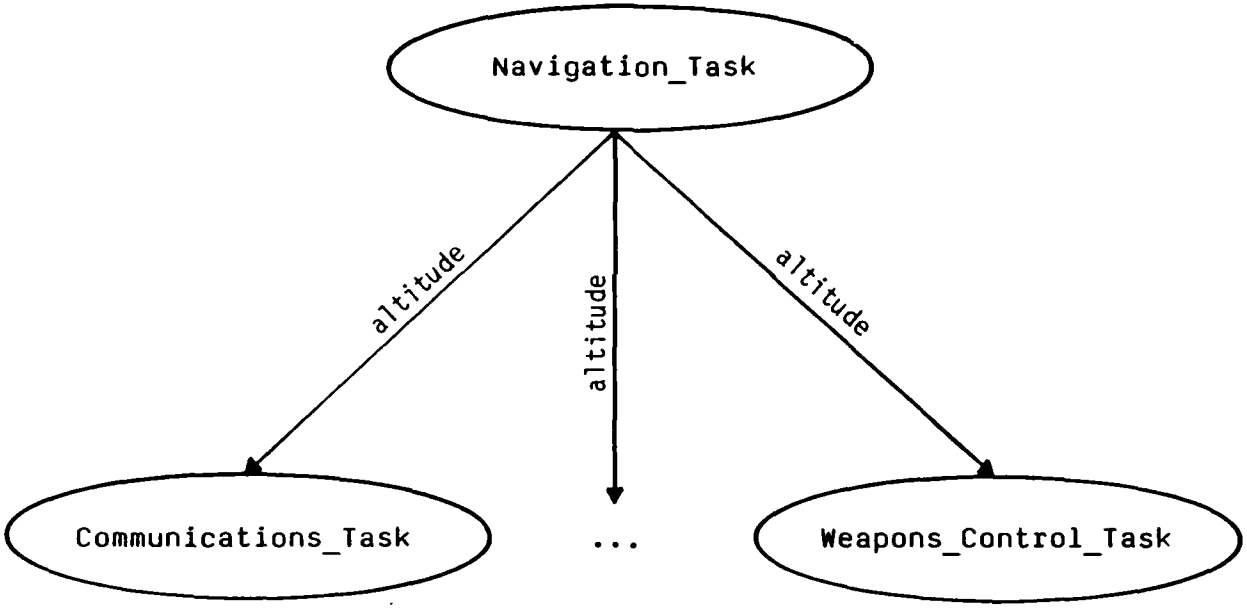
- IN THIS SECTION WE EXAMINE THE USE OF SHARED VARIABLES FOR TUNING.

INSTRUCTOR NOTES

- A NAVIGATION TASK DETERMINES THE CURRENT ALTITUDE AND MAKES IT AVAILABLE TO OTHER TASKS THAT NEED IT. FOR EXAMPLE, A COMMUNICATION TASK WOULD NEED THE ALTITUDE TO AIM ITS ANTENNAE OR A WEAPONS CONTROL TASK MIGHT NEED IT FOR FIRING ITS WEAPONS.

- OTHER TASKS MIGHT ALSO NEED THE ALTITUDE TO MONITOR CABIN PRESSURE, RADAR TASKS, ETC.

SHARING VARIABLES



- AN ONBOARD NAVIGATION SYSTEM MAY CONTAIN SEVERAL TASKS NEEDING THE Current_Altitude.

INSTRUCTOR NOTES

- MONITORS WERE DISCUSSED IN SECTION 12, SO THIS IS NOTHING NEW TO THE CLASS.
- FOR INSTRUCTOR ONLY.
  - TO MOTIVATE THE NEED FOR THE SHARED PRAGMA, WE START OFF WITH A MONITOR SOLUTION.
  - WE THEN ASSUME THAT A PROFILE IS OBTAINED THAT SHOWS THAT THE MONITOR SOLUTION IS TOO INEFFICIENT.
  - WE SHOW THAT THE MONITOR IS NOT REALLY NEEDED HERE, AND PRESENT A SHARED VARIABLE SOLUTION.
  - AFTER PRESENTING THE SOLUTION WE DISCUSS THE Shared PRAGMA.

THE MONITOR APPROACH

- THE TASKS CAN USE A MONITOR TO ACCESS THE CURRENT ALTITUDE.

```
task Altitude_Monitor is
   entry Deliver (New_Altitude : in Altitude_Type);
   entry Obtain (Altitude : out Altitude_Type);
end Altitude_Monitor;

task body Altitude_Monitor is
   Current_Altitude : Altitude_Type;
begin -- Altitude_Monitor
   accept Deliver (New_Altitude : in Altitude_Type) do
       Current_Altitude := New_Altitude;
   end Deliver;
   loop
       select
          accept Deliver (New_Altitude : in Altitude_Type) do
              Current_Altitude := New_Altitude;
          end Deliver;
       or
          accept Obtain (Altitude : out Altitude_Type) do
              Altitude := Current_Altitude;
          end Obtain;
       end select;
   end loop;
end Altitude_Monitor;
```

- THROUGH THE MONITOR, THEY SHARE THE VALUE OF THE Current_Altitude VARIABLE.
- ACCESS TO THE CURRENT ALTITUDE IS GRANTED IN A SAFE, CONTROLLED MANNER.

INSTRUCTOR NOTES

- POINT OUT THAT
  - EACH TIME THE Navigation_Task WANTS TO UPDATE THE CURRENT ALTITUDE, IT MUST ENTER RENDEZVOUS.
  - EACH TIME A TASK WANTS TO READ THE CURRENT ALTITUDE IT MUST ENTER RENDEZVOUS.
- Calculate_Altitude CALCULATES A NEW VALUE FOR THE CURRENT ALTITUDE.
- Adjust_Antenna_For AND Adjust_Trajectory_For SIMPLY READ THE CURRENT ALTITUDE VALUE.
- THE CONSTANT Adjustment_Factor IS ONLY USED TO ILLUSTRATE A PROBLEM IN SLIDE 22-9.

```
task body Navigation_Task is
      ...
   Navigation_Altitude : Altitude_Type;
begin
   loop
      Calculate_Altitude (Navigation_Altitude);
      Altitude_Monitor.Deliver (New_Altitude => Navigation_Altitude);
      ...
   end loop;
end Navigation_Task;

task body Communications_Task is
      ...
   Adjustment_Factor : constant := 1.7;
   Communications_Altitude : Altitude_Type;
begin
   loop
      ...
      Altitude_Monitor.Obtain (Altitude => Altitude_Value);
      Adjust_Antenna_For (Communications_Altitude * Adjustment_Factor);
      ...
   end loop;
end Communications_Task;

task body Weapons_Control_Task is
      ...
   Weapons_Altitude : Altitude_Type;
begin
   loop
      ...
      Altitude_Monitor.Obtain (Altitude => Weapons_Altitude);
      Adjust_Trajectory_For (Weapons_Altitude);
      ...
   end loop;
end Weapons_Control_Task;
```

INSTRUCTOR NOTES

- EMPHASIZE THAT PROFILE IS USED TO FIND THE "HOT SPOTS."

- BULLET 2. PROPER ANALYSIS OF ENTRY CALLS REQUIRES THAT WE LOOK AT HOW OFTEN THE ENTRY CALL IS REACHED AND HOW MUCH TIME IT SPENDS MAKING ENTRY CALLS.

THE RESULTS OF A PROFILE

- A PROFILE IS OBTAINED FOR THE SYSTEM.
- BASED ON THE PROFILE.
  - PERCENTAGE OF TIME SPENT IN ENTRY CALL STATEMENTS IS OUT OF PROPORTION TO FREQUENCY WITH WHICH STATEMENTS ARE REACHED.
  - CONCLUDE TOO MUCH TIME SPENT WAITING AT THE ENTRY CALL STATEMENTS.
- "HOT SPOTS" ARE PARTS OF THE PROGRAM USING CURRENT ALTITUDE.
  - UPDATING THE CURRENT ALTITUDE
    - Navigation_Task
  - READING THE CURRENT ALTITUDE.
    - Communications_Task
    - Weapons_Control_Task
    - etc.

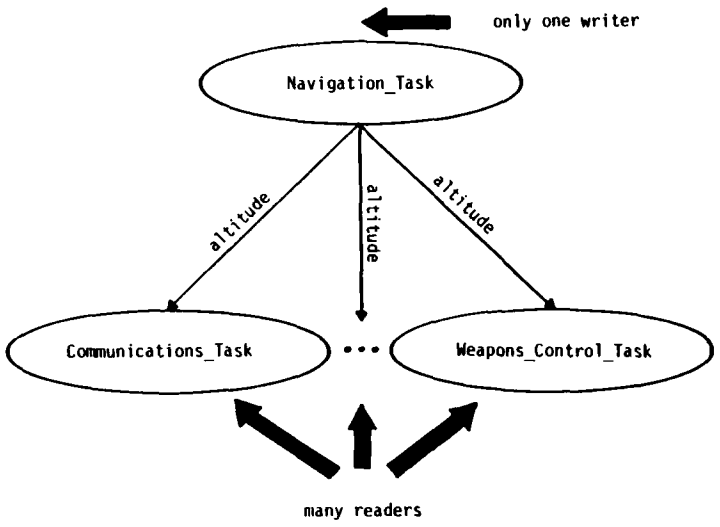
INSTRUCTOR NOTES

- THE OVERHEAD IN THE RENDEZVOUS IS UNNECESSARY FOR THIS SIMPLE DATA SHARING. REMEMBER BENTLEY'S REMARKS ABOUT BEAUTIFULLY STRUCTURED BUT TERRIBLY INEFFICIENT PROGRAMS.

- IT WAS PROPER TO USE THE MONITOR APPROACH IN SOLVING THE PROBLEM. WE LET THE PROFILE TELL US THAT IT IS NOT EFFICIENT. WE WILL TUNE THE PROGRAM TO USE Shared VARIABLES.

- EMPHASIZE THAT IT IS THE PROFILE THAT LEAD US TO CONCLUDE THAT THE MONITOR WAS A PERFORMANCE PROBLEM. WE ARE NOT IN GENERAL SAYING THAT MONITORS (OR RENDEZVOUS) ARE INEFFICIENT. SOME SYSTEMS MIGHT BE ABLE TO REMOVE MUCH OF THE OVERHEAD.

ANALYZING THE MONITOR SOLUTION



- THE MONITOR SOLUTION
  - FORCES READERS TO QUEUE UP TO READ Current_Altitude.
  - BUT ANALYSIS SHOWED THIS MONITOR CANNOT HANDLE THAT MUCH TRAFFIC QUICKLY ENOUGH.
- THE MONITOR IS PROVIDING PROTECTION AGAINST SIMULTANEOUS UPDATE.
  - NO SIMULTANEOUS UPDATE
    - ONLY WRITER IS Navigation_Task.
    - MANY READERS, BUT THEY CAN PROCEED IN PARALLEL.
  - SIMPLE SHARING OF VARIABLES WILL SUFFICE.

INSTRUCTOR NOTES

- EXPLAIN TO THE CLASS THAT NOW
  - Navigation_Task WRITES DIRECTLY TO Current_Altitude.
  - Communications_Task AND Weapons_Control_Task DIRECTLY READ Current_Altitude.
- WE HAVE REMOVED ALL OVERHEAD ASSOCIATED WITH THE MONITOR. STRESS THAT THIS COULD NOT SENSIBLY BE DONE WITH TWO OR MORE WRITERS, DUE TO THE SIMULTANEOUS UPDATE PROBLEM.
- THE NEXT SLIDE EXPLAINS WHY WE NEED THE Shared PRAGMA.

```
Current_Altitude : Altitude_Type;
pragma Shared (Current_Altitude);
        ...
task body Navigation_Task is
        ...
begin
   loop
        Calculate_Altitude (Current_Altitude); -- Set Current_Altitude
        ...
   end loop;
end Navigation_Task;
task body Communications_Task is
        ...
   Adjustment_Factor : constant := 1.7;
begin
   loop
        ...
        Adjust_Antenna_For (Current_Altitude * Adjustment_Factor);
                                        -- Examine Current_Altitude
        ...
   end loop;
end Communications_Task;
task body Weapons_Control_Task is
        ...
begin
   loop
        ...
        Adjust_Trajectory_For (Current_Altitude); -- Examine Current_Altitude
        ...
   end loop;
end Weapons_Control_Task;
```

INSTRUCTOR NOTES

WHY THE Shared PRAGMA IS NEEDED

- A COMPILER IS ALLOWED TO OPTIMIZE A TASK'S REFERENCES TO A SHARED VARIABLE IN A SCALAR TYPE OR ACCESS TYPE BY KEEPING A LOCAL COPY OF IT (POSSIBLY IN A REGISTER). FOR THIS TASK, ACCESSING THE LOCAL COPY IS EQUIVALENT TO ACCESSING THE SHARED VARIABLE.

- IN THE ABSENCE OF THE Shared PRAGMA, THE SHARED COPY AND THE TASK'S LOCAL COPY DO NOT NECESSARILY AGREE, EXCEPT AT THE START OF THE TASK, END OF THE TASK OR DURING A RENDEZVOUS.

- THE Shared PRAGMA ENSURES BOTH COPIES ARE THE SAME.

  pragma Shared (identifier);

  WHERE identifier IS THE NAME OF A DECLARED VARIABLE.

- THE PRAGMA IS NOT ALLOWED FOR RECORDS OR ARRAYS, BUT IT IS NOT NEEDED FOR THEM.

- THE PRAGMA MUST OCCUR AFTER THE DECLARATION.

INSTRUCTOR NOTES

- THE CONSTANT Adjustment_Factor IS USED TO SHOW HOW A COMPILER MIGHT USE A SHARED VARIABLE.

- IN THE Communications_Task, WE SHOW SOME POSSIBLE PSEUDO ASSEMBLER CODE TO ILLUSTRATE WHAT MIGHT HAPPEN IN THE ABSENCE OF THE Shared PRAGMA.

  - JUST BEFORE ENTERING THE LOOP, THE VALUE
    Current_Altitude * Adjustment_Factor
    IS PLACED INTO REGISTER Reg.
  - THE VALUE IN Reg IS NOT MODIFIED DURING THE LOOP.
  - WHEN THE CALL TO Adjust_Antenna_For IS MADE; THE VALUE OF Reg IS STORED IN Parameter.

- THE COMPILER PERFORMED THIS OPTIMIZATION BECAUSE

  - IT FOUND THAT THE VALUE OF Current_Altitude WAS CONSTANT THROUGHOUT THE LOOP.
  - SINCE Current_Altitude IS CONSTANT THROUGHOUT THE LOOP, SO IS
    Current_Altitude * Adjustment_Factor.
  - MOVING THE EXPRESSION OUTSIDE OF THE LOOP ALLOWS THIS "CONSTANT" EXPRESSION TO BE EVALUATED ONLY ONCE.

WHY THE SHARED PRAGMA IS NEEDED - CONTINUED

- IN THE ABSENCE OF THE Shared PRAGMA A "SMART" COMPILER MAY USE THE OPTIMIZATION CALLED CODE MOTION TO MOVE LOADING A REGISTER WITH Current_Altitude OUT OF THE LOOP.

```
Current_Altitude : Altitude_Type;
        ...
task body Navigation_Task is
        ...
begin
   loop
        Calculate_Altitude (Current_Altitude); -- Set Current_Altitude
        ...
   end loop;
end Navigation_Task;
task body Communications_Task is
        ...
   Adjustment_Factor : constant := 1.7;
begin
LOAD Reg, Current_Altitude
MUL Reg, Adjustment_Factor  ➡  loop
        ...
STORE Reg, Parameter
CALL Adjust_Antenna_For     ➡      Adjust_Antenna_For (Current_Altitude * Adjustment_Factor);
                                   -- Evaluate Current_Altitude
        ...
   end loop;
end Communications_Task;
task body Weapons_Control_Task is
        ...
begin
   loop
...
        Adjust_Trajectory_For (Current_Altitude);
                               -- Evaluate Current_Altitude
        ...
   end loop;
end Weapons_Control_Task;
```

INSTRUCTOR NOTES

- ALLOW 45 MINUTES FOR THIS SECTION.

Section 23

MINIMIZING BLOCKING

INSTRUCTOR NOTES

- BULLET 3:

  "FAIR" DEPENDS ON THE RESOURCE.

MAXIMIZING THROUGHPUT

- IF TASKS SPEND TOO MUCH OF THEIR TIME WAITING FOR OTHER TASKS OR DEVICES TO COMPLETE CERTAIN ACTIONS, THEN PROCESSORS MAY BE UNDERUTILIZED.

- TASKS CAN BE REWRITTEN
  - TO REDUCE AMOUNT OF TIME SPENT IDLE, AND
  - TO MAXIMIZE POTENTIAL FOR PARALLELISM

- SCHEDULING THE USE OF RESOURCES CAN BE TO DISTRIBUTE THE TIME SPENT WAITING FOR RESOURCES IN A "FAIR" MANNER.

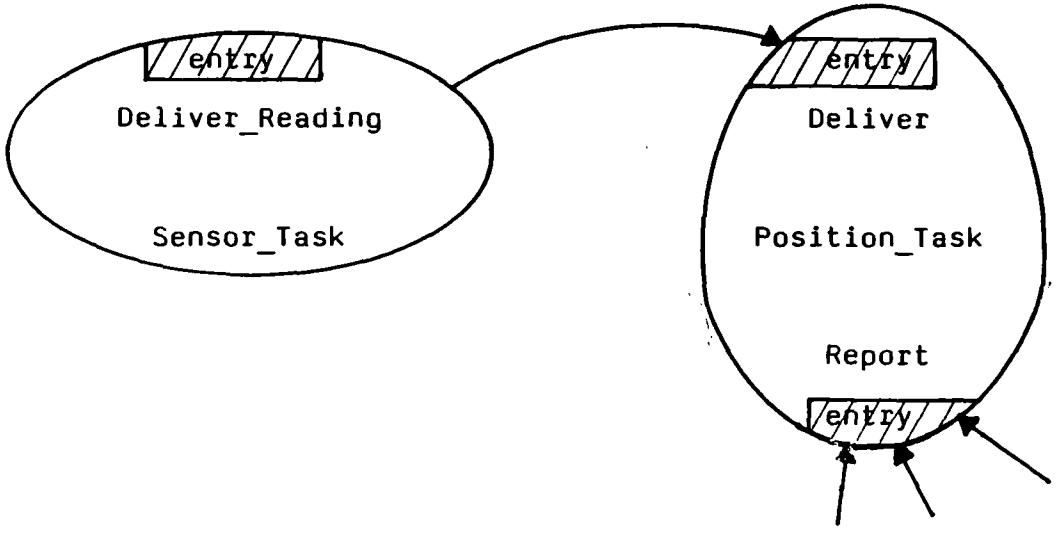
INSTRUCTOR NOTES

- THIS EXAMPLE IS SIMILAR TO THE NAVIGATION EXERCISE, SO IT SHOULD NOT NEED MUCH EXPLANATION.

- TWO VERSIONS WILL BE PRESENTED. THE FIRST WILL PERFORM EXCESSIVE COMPUTATION WITHIN THE ACCEPT STATEMENT. THE SECOND VERSION WILL CORRECT THIS.

COMPUTATIONS WITHIN ACCEPT STATEMENTS

- AS PART OF A NAVIGATION SYSTEM
  - A Sensor_Task DELIVERS AVERAGED SENSOR READINGS TO A Position_Task BY CALLING THE Deliver ENTRY.
  - THE Position_Task USES THE AVERAGED SENSOR READING AND THE PREVIOUS POSITION TO CALCULATE THE CURRENT POSITION.
  - OTHER TASKS OBTAIN THE CURRENT POSITION BY CALLING THE Report ENTRY.



- IF THE RENDEZVOUS TAKES "TOO LONG" Sensor_Task MAY MISS THE NEXT READING.

INSTRUCTOR NOTES

- THE Sensor_Task CALLS THE Deliver ENTRY OF THE Position_Task TO DELIVER THE AVERAGED READING AND THEN WAITS WHILE AN UPDATED POSITION IS BEING CALCULATED. WHILE IT IS WAITING, IT MAY LOSE A READING.

TOO MUCH COMPUTATION WITHIN ACCEPT STATEMENTS

```
task Sensor_Task is
   entry Deliver_Reading (Reading : in Float);
end Sensor_Task;

task body Sensor_Task is
      ...
   loop
      Sum := 0.0;
      for I in 1 .. 5 loop
         accept Deliver_Reading (Reading : in Float) do
            Sum := Sum + Reading;
         end Deliver_Reading;
      end loop;
      Position_Task.Deliver (Sum / 5.0);
   end loop;
end Sensor_Task;

task Position_Task is
   entry Deliver (Averaged_Reading : in Float);
   entry Report (Position : out Position_Type);
end Position_Task;

task body Position_Task is
      ...
   loop
      select
         accept Deliver (Averaged_Reading : in Float) do
            Previous_Position :=
                Updated_Position (Previous_Position, Averaged_Reading);
         end Deliver;
      or
         accept Report (Position : out Position_Type) do
            Position := Previous_Position;
         end Report;
      end select;
   end loop;
end Position_Task;
```

Sensor_Task waits

INSTRUCTOR NOTES

- THIS VERSION SHOWS THAT THE Update_Position CALCULATION SHOULD BE REMOVED FROM THE accept STATEMENT. ALL THAT THE BODY OF THE ACCEPT STATEMENT DOES IS TO COPY THE INPUT PARAMETER.

KEEP ACCEPT STATEMENTS SHORT

- THE SELECT STATEMENT SHOULD BE REWRITTEN AS:

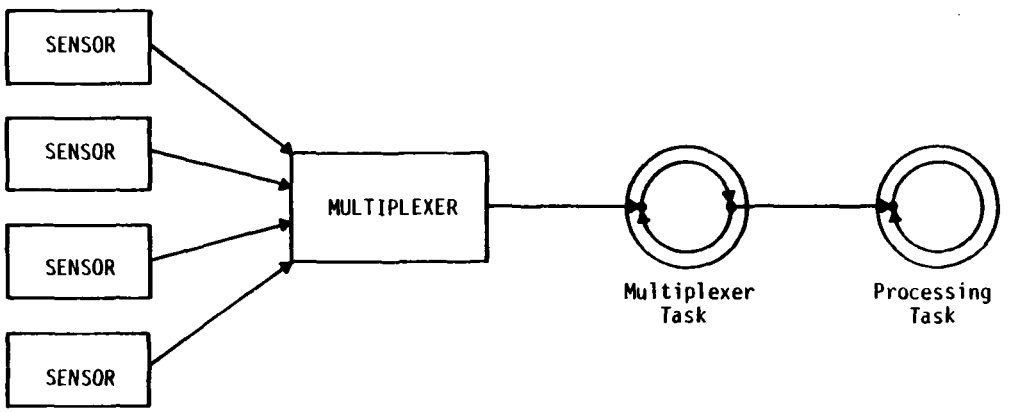
```
loop
   select
      accept Deliver (Averaged_Reading : in Float) do
         New_Average := Averaged_Reading;
      end Deliver;
      Previous_Position := Updated_Position (Previous_Position,
      New_Average);
   or
      accept Report (Position : out Position_Type) do
         Position := Previous_Position;
      end Report;
   end select;
end loop;
```

- KEEP ACCEPT STATEMENTS SHORT.
  - MOST ACTIVITIES DO NOT NEED TO OCCUR DURING RENDEZVOUS.

INSTRUCTOR NOTES

- THIS IS A REVIEW OF THE MULTIPLEXER TASK FROM THE MESSAGE BUFFER SECTION.
- THE PURPOSE OF THIS SLIDE AND THE NEXT IS JUST TO REVIEW BUFFERING AS A WAY OF INCREASING PARALLELISM BY LETTING THE BUFFER DO THE WAITING.
- IN THE TIME LINES, THE UP ARROWS SHOW WHEN THE MULTIPLEXER TASK DELIVERS THE PACKET TO THE PROCESSING TASK. THE DOWN ARROW SHOWS WHEN THE MULTIPLEXER TASK READS THE MULTIPLEXER PACKET. THE MULTIPLEXER TIME LINE IS DIVIDED INTO 200 MILLISECOND INTERVALS.
- POINT OUT WHERE THE PACKET IS LOST.

MULTIPLEXER_TASK

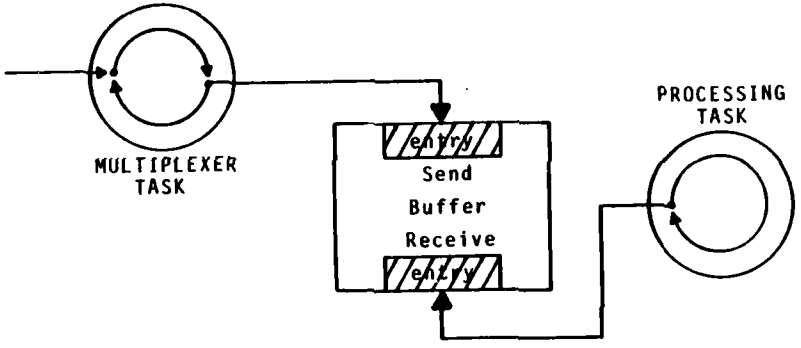


- EVERY 200 MILLISECONDS A MULTIPLEXER ASSEMBLES A PACKET OF 4 SENSOR READINGS.
- THE MULTIPLEXER TASK READS FROM THE MULTIPLEXER EVERY 200 MILLISECONDS AND DELIVERS THE PACKET TO THE PROCESSING TASK.
- NORMALLY, A PACKET CAN BE PROCESSED IN UNDER 200 MILLISECONDS.
- PROBLEM
  - OCCASIONALLY, PROCESSING OF A PACKET TAKES A LITTLE LONGER THAN 200 MILLISECONDS.
  - RESULTS IN MULTIPLEXER TASK BEING DELAYED FOR MORE THAN 200 MILLISECONDS.
  - IF MULTIPLEXER PACKET NOT READ WITHIN 200 MILLISECONDS, IT IS LOST.

INSTRUCTOR NOTES

BUFFERING

- PROBLEM SOLVED BY USING A BUFFER



- AS LONG AS AVERAGE PROCESSING TIME IS UNDER 200 MILLISECONDS AND PROCESSING TASK DOES NOT DEVIATE MUCH FROM THIS AVERAGE, THE PROCESSING TASK DOES NOT SLOW DOWN THE MULTIPLEXER TASK.

- PARALLELISM IS IMPROVED BY LETTING THE BUFFER DO THE WAITING.

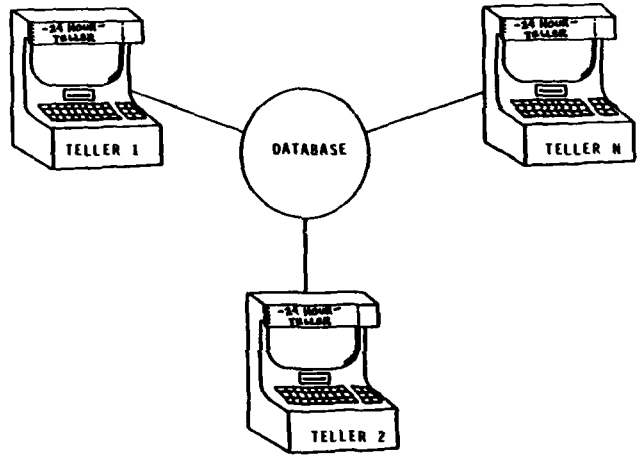
INSTRUCTOR NOTES

- THE TWO POLICIES REPRESENT THE TWO EXTREMES OF LOCKING.

- BY LOCKING THE ENTIRE DATABASE OTHER TELLERS ARE PREVENTED FROM ACCOMPLISHING ANY USEFUL WORK. SINCE LOCKING IS ONLY NEEDED AT THE ACCOUNT LEVEL, THIS IS WHERE IT SHOULD BE PERFORMED.

AN AUTOMATIC TELLER SYSTEM

- AN AUTOMATIC TELLER SYSTEM HAS AN AUTOMATIC TELLER AT MANY LOCATIONS. THEY SHARE A COMMON CUSTOMER DATABASE.



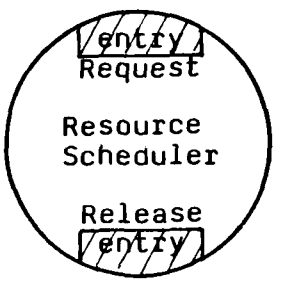
- IN ORDER TO PERFORM A TRANSACTION, AN AUTOMATIC TELLER CAN EITHER
  - LOCK THE ENTIRE DATABASE
    - ONLY ONE TRANSACTION AT A TIME
    - CUSTOMERS GET SLOW SERVICE
  - LOCK A CUSTOMER'S RECORD WHEN NEEDED
    - SEVERAL TRANSACTIONS MAY PROCEED IN PARALLEL
    - CUSTOMERS GET FASTER SERVICE

INSTRUCTOR NOTES

- THE RESOURCE SCHEDULER QUEUE IS A DATA STRUCTURE WITHIN THE SCHEDULER.

SCHEDULING THE USE OF RESOURCES

- RESOURCES BEING SHARED AMONG SEVERAL TASKS



Resource
Scheduler
Queue

T2	T5	T1	...

- FOR EACH RESOURCE TYPE THERE IS A RESOURCE SCHEDULER THAT SCHEDULES USE OF THE RESOURCES.
- THE SCHEDULER DEPENDS ON A
  - POLICY FOR SCHEDULING
  - MECHANISM FOR IMPLEMENTING THE POLICY
- IF, WHEN A TASK REQUESTS RESOURCES IN A RESOURCE TYPE, THE RESOURCES ARE AVAILABLE, THE TASK IS GIVEN THE RESOURCES; OTHERWISE THE TASK IS PLACED ON A QUEUE ACCORDING TO THE POLICY.
- WHEN A TASK RELEASES RESOURCES IN A RESOURCE TYPE, THE TASK AT THE HEAD OF THE QUEUE IS ALLOWED TO PROCEED IF THERE ARE NOW ENOUGH RESOURCES.

INSTRUCTOR NOTES

- OTHER POLICIES CAN BE FORMED BY USING A COMBINATION OF THESE POLICIES. FOR EXAMPLE, MANY TASKS CAN BE DIVIDED INTO A SMALL NUMBER OF PRECEDENCE LEVELS. WITHIN A PRECEDENCE LEVEL ONE OF THE OTHER POLICIES, SAY HIGHEST RESPONSE RATIO NEXT, MAY BE USED IN EACH OF THE PRECEDENCE LEVELS.

SCHEDULING POLICY

- FIRST COME, FIRST SERVED
  - TASKS EXECUTED IN ORDER OF ARRIVAL
  - PREVENTS ANY TASK FROM WAITING "TOO" LONG.
  - TREATS ALL TASKS EQUALLY FAIRLY.
  - USED IN Ada FOR QUEUING CALLS TO AN ENTRY.

- SHORTEST SERVICE TIME FIRST
  - SERVICE TIME: ESTIMATE OF HOW LONG TASK WILL USE RESOURCE
  - TASK MAY GIVE SERVICE TIME AT REQUEST, OR MAY BE DETERMINED DYNAMICALLY BASED ON AVERAGE OF PREVIOUS USE.
  - FAVORS TASKS THAT WILL USE RESOURCE AND RETURN IT FAST
  - TASKS NEEDING A RESOURCE FOR A LONG TIME ARE PENALIZED.

- HIGHEST RESPONSE RATIO NEXT
  - WAITING TIME: HOW LONG TASK HAS BEEN WAITING FOR RESOURCE
  - RESPONSE RATIO:

$$\frac{\text{WAITING TIME} + \text{SERVICE TIME}}{\text{SERVICE TIME}}$$

    MEASURES DEGRADATION IN TASK'S EXECUTION SPEED DUE TO "PRESENCE" OF OTHER TASKS.
  - TASK WITH HIGHER RESPONSE TIME SCHEDULED NEXT.
  - BALANCE BETWEEN PREVIOUS POLICIES.
    - FAVORS TASKS WITH SHORTER SERVICE TIME AS DOES SHORTEST SERVICE TIME FIRST
    - LIMITS WAITING TIME OF LONGER TASKS AS DOES FIRST COME, FIRST SERVED.

- HIGHEST PRECEDENCE NEXT
  - EACH TASK ASSIGNED A PRECEDENCE LEVEL AFTER ACTIVATION.
  - SEVERAL TASKS MAY BE AT SAME PRECEDENCE LEVEL.

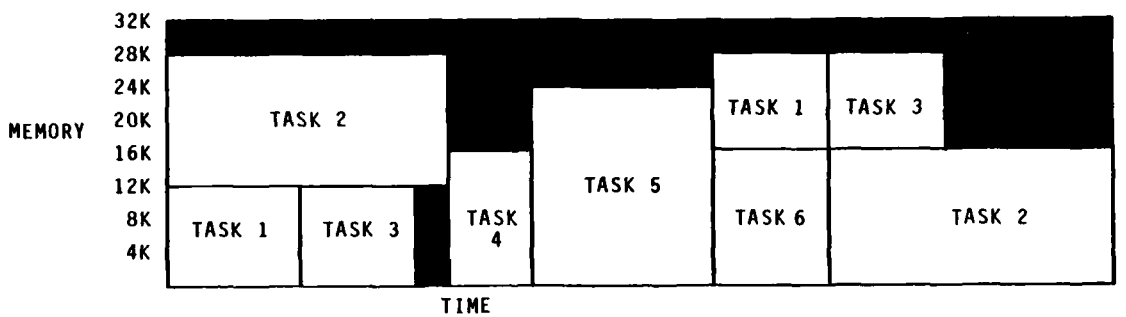
INSTRUCTOR NOTES

- THE TIME MAP SHOWS HOW MEMORY USAGE VARIES WITH TIME. THE FILLED PORTION OF THE MAP INDICATES UNUSED MEMORY.

- TASK 1 IS FIRST GIVEN ITS MEMORY. SINCE THERE IS ENOUGH MEMORY FOR TASK 2 TO EXECUTE, IT IS ALLOWED TO DO SO. TASK 3 NEEDS 12K BUT THERE IS ONLY 4K AVAILABLE - IT MUST WAIT. WHEN TASK 1 FINISHES ITS ACTIVITY, IT RETURNS ITS 12K, WHICH ALLOWS TASK 3 TO START ITS ACTIVITY. TASK 4 NEEDS 16K, BUT THERE IS ONLY 4K AVAILABLE SO IT MUST WAIT. WHEN TASK 3 FINISHES ITS ACTIVITY, IT RETURNS ITS MEMORY. THERE IS NOW 16K AVAILABLE, BUT TASK 4 NEEDS 16K OF CONTIGUOUS MEMORY, SO IT STILL WAITS. FINALLY, TASK 2 FINISHES ITS ACTIVITY, WHICH ALLOWS TASK 4 TO START ITS ACTIVITY. (POINT OUT THAT WHEN TASK 1 STARTS ITS ACTIVITY AGAIN, IT DOES NOT GET THE SAME 12K.)

SCHEDULING USE OF MEMORY

- SEVERAL CYCLIC ACTIVITIES EACH HAVE MEMORY REQUIREMENTS.
  - THE MEMORY REQUIREMENT OF EACH ACTIVITY IS SPECIFIED WHEN THE EXECUTIVE IS BEING STARTED UP.
  - THE NEXT SCHEDULED ACTIVITY IS ALLOWED TO BEGIN ONCE ITS MEMORY REQUIREMENTS CAN BE SATISFIED.
  - MEMORY ALLOCATED MUST BE CONTIGUOUS.
- A TIME MAP OF MEMORY USAGE IN A 32K SYSTEM MIGHT LOOK LIKE



TASK	MEMORY NEEDED
1	12K
2	16K
3	12K
4	16K
5	24K
6	16K

INSTRUCTOR NOTES

- THIS SLIDE SHOWS THE IMPLEMENTATION OF THE SCHEDULER DESCRIBED ON THE PREVIOUS SLIDE. ENTRY FAMILIES ARE USED. DO NOT SPEND MUCH TIME ON THIS SLIDE.

- WHEN THE SYSTEM IS STARTED UP, EACH TASK CALLS THE CORRESPONDING Memory_Needed ENTRY IN THE FAMILY TO SPECIFY HOW MUCH MEMORY IT WILL NEED FOR ITS ACTIVITY. SUBSEQUENTLY, WHEN A TASK WANTS TO START ITS ACTIVITY IT CALLS THE CORRESPONDING Get_Memory ENTRY. IF THE CALL TO Enough_Memory RETURNS True, THEN THERE IS ENOUGH CONTIGUOUS MEMORY FOR THE TASK TO START THE ACTIVITY, SO THE GUARD EVALUATES TO True, LETTING THE Get_Memory ENTRY BE ACCEPTED. OTHERWISE ONLY THE Release ENTRY CAN BE ACCEPTED.

- IF THE Get_Message ENTRY IS ACCEPTED, MEMORY IS ALLOCATED BY CALLING Allocate_Memory. THIS ALLOCATES MEMORY FOR THE TASK, AND KEEPS TRACK OF WHAT BLOCKS HAVE BEEN ALLOCATED.

- IF THE Release ENTRY IS ACCEPTED, THEN Free_Memory IS CALLED TO REFLECT THE DEALLOCATION OF THE MEMORY.

- THE Allocate_Memory PROCEDURE CALL MUST BE INSIDE THE accept STATEMENT BECAUSE THE ENTIRE COMPUTATION SHOULD TAKE PLACE INSIDE THE RENDEZVOUS. THE PROCEDURE CALL Free_Memory SAVES SPACE ON THE SLIDE! THE Id COULD BE COPIED TO A LOCAL VARIABLE INSIDE THE RENDEZVOUS AND THE PROCEDURE CALL MOVED AFTER THE RENDEZVOUS.

A HIGHEST PRECEDENCE FIRST BASED MEMORY SCHEDULER

```
task Memory_Scheduler is
   entry Memory_Needed (Task_Id_Range) (Amount : in Natural);
   entry Get_Memory (Task_Id_Range);
   entry Release (Id : in Task_Id_Range);
end Memory_Scheduler;

task body Memory_Scheduler is

   Task_Id     : Task_Id_Range := 1;
   Memory_Size : array (Task_Id_Range) of Natural;

   function Enough_Memory (Amount : Natural) return Boolean is separate;
   procedure Allocate_Memory (Id  : in Task_Id_Range) is separate;
   procedure Free_Memory (Id : in Task_Id_Range) is separate;

begin -- Memory_Scheduler
   for I in Task_Id_Range loop
      accept Memory_Needed (I) (Amount : in Natural) do
         Memory_Size (I) := Amount;
      end Memory_Needed;
   end loop;

   loop
      select
         when Enough_Memory (Memory_Size (Task_Id)) =>
            accept Get_Memory (Task_Id) do
               Allocate_Memory (Task_Id);
            end Get_Memory;
            if Task_Id /= Task_Id_Range'Last then
               Task_Id := Task_Id + 1;
            else
               Task_Id := 1;
            end if;
      or
            accept Release (Id : Task_Id_Range) do
               Free_Memory (Id);
            end Release;
      end select;
   end loop;
end Memory_Scheduler;
```

INSTRUCTOR NOTES

OPTIMAL SCHEDULING

IN ORDER TO PROVIDE 'OPTIONAL' SCHEDULING FOR A REAL TIME SYSTEM, A SCHEDULER MUST KNOW ABOUT APPLICATION CODE AND ITS REQUIREMENTS.

OPTIMAL SCHEDULING IN THIS CASE MEANS THAT GIVEN A SET OF TASKS, EACH OF WHICH HAS A DEADLINE, AN OPTIMAL SCHEDULER WILL SCHEDULE THE TASKS SUCH THAT EACH WILL MEET ITS DEADLINE, PROVIDED THERE EXISTS A SCHEDULE WHERE THIS IS POSSIBLE.

FOR A SINGLE RESOURCE (SUCH AS A SINGLE PROCESSOR) THE OPTIMAL SCHEDULER IS CALLED AN EARLIEST DEADLINE SCHEDULER. THIS SCHEDULER ALWAYS EXECUTES THE TASK WITH THE EARLIEST DEADLINE.

THIS TYPE OF SCHEDULER OBVIOUSLY NEEDS TO KNOW THE DEADLINES ASSOCIATED WITH EACH TASK.

INSTRUCTOR NOTES

SCHEDULING CONSIDERATIONS

WHEN THERE IS MORE THAN ONE RESOURCE, THE PROBLEM BECOMES MORE COMPLEX. THE EARLIEST DEADLINE SCHEDULER NO LONGER PROVIDES OPTIMAL SCHEDULING IN ALL CASES. CONSIDER THE CASE OF TWO PROCESSORS WITH THREE TASKS.

TASK 1: 1 SEC OF PROCESSING DEADLINE IN 2 SEC

TASK 2: 1 SEC OF PROCESSING DEADLINE IN 1 SEC

TASK 3: 3 SEC OF PROCESSING DEADLINE IN 3 SEC

THE EARLIEST DEADLINE WOULD SCHEDULE TASKS 1 AND 2 TO START IMMEDIATELY, THEREFORE TASK 3 WOULD MISS ITS DEADLINE. THERE IS, HOWEVER, A POSSIBLE SCHEDULE THAT MEETS ALL DEADLINES -- TASKS 2 AND 3 ARE STARTED IMMEDIATELY AND TASK 1 GETS STARTED WHEN TASK 2 COMPLETES.

INSTRUCTOR NOTES

DON'T GET INTO A DISCUSSION OF SYSTEMS WITH MORE THAN TWO RESOURCES. THIS CAN BE A VERY COMPLEX TOPIC. IF SOMEONE ASKS, REFER THEM TO THE PAPER BY MOK AND DERTOUZOS.

OPTIMAL SCHEDULER

THE OPTIMAL SCHEDULER FOR TWO RESOURCES IS CALLED THE LEAST LEXITY SCHEDULER. THIS SCHEDULER RUNS THE PROGRAMS THAT HAVE THE LEAST DIFFERENCE BETWEEN THEIR REMAINING RUNTIME AND THEIR DEADLINE. THIS SCHEDULER THEN HAS TO KNOW BOTH THE DEADLINES FOR EACH TASK AND THE RUNTIME FOR EACH TASK.

EVEN THIS SCHEDULER IS NOT NECESSARILY OPTIMAL FOR SITUATIONS WITH MORE THAN TWO RESOURCES. THERE IS NO KNOWN GENERAL OPTIMAL SCHEDULER FOR AN ARBITRARY NUMBER OF PROCESSORS. THE PROBLEMS IN TRYING TO DEFINE ONE ARE DISCUSSED IN A PAPER BY AL MOK WHICH IS LISTED IN THE BIBLIOGRAPHY.

INSTRUCTOR NOTES

ALLOW 50 MINUTES FOR THIS SECTION, NOT INCLUDING THE EXERCISE. ALLOW 70 MINUTES FOR THE EXERCISE (APPROXIMATELY 40 MINUTES BEFORE THE LUNCH BREAK AND 30 MINUTES AFTERWARDS).

Section 24

MERGING TASKS

INSTRUCTOR NOTES

- BULLET 1: BESIDES THESE SPECIFIC COSTS, A COMPILER MAY BE UNABLE TO FIND AN OPPORTUNITY FOR OPTIMIZATION WHEN PARTS OF A COMPUTATION ARE SPLIT ACROSS DIFFERENT TASKS.
  - ITEM 1: THIS INCLUDES STORAGE FOR EACH TASK'S LOCAL DATA PLUS WHAT IS SOMETIMES CALLED A "TASK CONTROL BLOCK." A TASK CONTROL BLOCK TYPICALLY INCLUDES SPACE TO SAVE REGISTERS AND THE PROGRAM COUNTER, STORAGE FOR ENTRY QUEUES, AND OTHER CONTROL INFORMATION.
  - ITEM 2: THE "STATE OF A TASK" INCLUDES REGISTERS AND PROGRAM COUNTER, FOR EXAMPLE.
  - ITEM 3: THE QUEUE MUST BE MANIPULATED WHEN AN ENTRY IS CALLED, WHEN A CALL IS ACCEPTED, AND WHEN A CALL IS CANCELLED.
  - ITEM 4: AFTER A RENDEZVOUS, BOTH PARTICIPANTS ARE ELIGIBLE TO EXECUTE AND ONE MUST BE SELECTED. THIS MAY REQUIRE EXAMINING PRIORITIES.
  - ITEM 5: DEPENDING ON THE IMPLEMENTATION, THE TIME TO CHOOSE THE NEXT TASK FOR EXECUTION MAY INCREASE WITH THE NUMBER OF TASKS.
  - ITEM 6: IN THE CASE OF in PARAMETERS, FOR EXAMPLE, ACTUAL PARAMETERS MUST GENERALLY BE COPIED TO FORMAL PARAMETERS, AND FORMAL PARAMETERS MUST BE COPIED TO LOCAL VARIABLES IN THE ACCEPTING TASK.
- BULLET 2: THIS SECTION ONLY SHOWS HOW TO MERGE TWO TASKS.
- BULLET 4: THIS RE-EMPHASIZES THE POINT MADE IN SECTION 21.
  - ITEM 2: BULLET 1 LISTED POTENTIAL COSTS OF A LARGE NUMBER OF TASKS. FOR SOME IMPLEMENTATIONS, THESE COSTS MAY BE INSIGNIFICANT.
  - ITEM 3: A CLUE IS A PROGRAM SPENDING AN UNUSUALLY LARGE PORTION OF ITS TIME EXECUTING SYSTEM ROUTINES.

POSSIBLE COSTS OF HAVING MANY TASKS

- DEPENDING ON THE IMPLEMENTATION OF TASKING IT MAY BE EXPENSIVE IN TERMS OF TIME AND SPACE TO HAVE A LARGE NUMBER OF TASKS.
  - STORAGE FOR TASK OBJECTS.
  - SAVING THE STATE OF ONE TASK AND RESTORING THE STATE OF ANOTHER WHEN SWITCHING TASKS.
  - MAINTAINING ENTRY QUEUES WHEN TASKS COMMUNICATE.
  - MAKING SCHEDULING DECISIONS AFTER TASKS COMMUNICATE.
  - SEARCH TIME TO FIND THE NEXT TASK TO EXECUTE.
  - EXTRA ASSIGNMENT STATEMENTS COPYING DATA INTO OR OUT OF accept STATEMENTS.
- THIS OVERHEAD CAN BE REDUCED BY MERGING TASKS.
  - TWO TASKS THAT COMMUNICATE THROUGH RENDEZVOUS ARE MERGED INTO A SINGLE TASK BY TRANSFORMING THE SOURCE PROGRAM ACCORDING TO WELL-DEFINED RULES.
  - IF THIS PROCESS IS REPEATED, AN ARBITRARY NUMBER OF TASKS CAN BE MERGED INTO A SINGLE TASK.
- THE MERGED TASK CAN BE SUBSTANTIALLY LESS UNDERSTANDABLE THAN THE TWO ORIGINAL TASKS.
- BEFORE MERGING TASKS:
  - ASCERTAIN THAT YOU REALLY HAVE A PERFORMANCE PROBLEM.
  - ASCERTAIN THAT THERE REALLY IS AN OVERHEAD ASSOCIATED WITH A LARGE NUMBER OF TASKS UNDER YOUR IMPLEMENTATION.
  - ASCERTAIN THAT THIS OVERHEAD REALLY IS THE SOURCE OF YOUR PERFORMANCE PROBLEM.

INSTRUCTOR NOTES

- BULLET 1:

  THESE CONDITIONS ALLOW US TO MAKE ASSUMPTIONS THAT SIMPLIFY THE PROBLEM OF MERGING TASKS. IN GENERAL, ANY GROUP OF TASKS CAN BE MERGED, PROVIDED THAT ENTRIES USED FOR COMMUNICATION WITHIN THE GROUP ARE NOT ALSO CALLED FROM OUTSIDE THE GROUP. HOWEVER, THE MANIPULATIONS FOR THE GENERAL CASE ARE TOO COMPLICATED TO BE SHOWN HERE.

- BULLET 3:

  THE REST OF THIS SECTION IS DEVOTED TO CONSIDERING FIRST THE SIMPLE CASE, THEN THE COMPLICATED CASE.

EASILY MERGEABLE TASKS

- TWO TASKS ARE EASILY MERGEABLE IF:
  - THEY COMMUNICATE DIRECTLY, THROUGH ENTRIES NOT CALLED BY ANY OTHER TASK.
  - FOR EACH ENTRY, ALL PARAMETERS ARE OF MODE in OR ALL ARE OF MODE out.
  - ALL accept STATEMENTS USED FOR COMMUNICATION BETWEEN THESE TASKS ARE STANDALONE accept STATEMENTS, NOT ALTERNATIVES IN A SELECTIVE WAIT.
  - THE ENTRY CALLS USED FOR COMMUNICATION BETWEEN THESE TASKS ARE ORDINARY ENTRY CALLS, NOT TIMED OR CONDITIONAL.

- TASK MERGING CAN BE APPROPRIATE WITH STREAM-ORIENTED TASK DESIGN.
  - LARGE NUMBER OF TASKS CREATED.
  - ALL ASSUMPTIONS LISTED ABOVE ARE VALID.

- TWO CASES TO CONSIDER:
  - THE SIMPLE CASE: ALL COMMUNICATION IS THROUGH A SINGLE ENTRY CALL AND A SINGLE accept STATEMENT.
  - THE COMPLICATED CASE: ONE OR MORE ENTRY CALL STATEMENTS, MAKING CALLS ACCEPTED BY ONE OR MORE accept STATEMENTS.

INSTRUCTOR NOTES

SLOW DOWN FOR THE SLIDE. IT CONTAINS IMPORTANT POINTS THAT WILL TAKE A WHILE TO EXPLAIN.

- BULLET 1: THE TRANSFORMATION DEPENDS ON WHICH TASK SENDS DATA TO WHICH, REGARDLESS OF THE DIRECTION OF THE RENDEZVOUS. (BULLET 1, ITEM 2 ON THE PREVIOUS SLIDE REQUIRED THAT ALL DATA GO IN THE SAME DIRECTION.) THE REST OF THIS SECTION REFERS TO SENDING AND RECEIVING DATA RATHER THAN CALLING OR ACCEPTING ENTRIES. THE "S" AND "R" IN THE FLOWCHART STAND FOR "SENDING TASK" AND "RECEIVING TASK."

  IMPORTANT: MAKE SURE THE CLASS UNDERSTANDS THAT CERTAIN BOXES IN THE FLOW CHART MAY CORRESPOND TO DOING NOTHING FOR GIVEN TASKS. FOR EXAMPLE, IF THE RECEIVING TASK DOES NOTHING BEFORE ENTERING ITS LOOP, THEN BOX R1 IS "NULL." IF BOTH TASKS EXECUTE NON-TERMINATING LOOPS, THEN S4, S6, R4, AND R6 ARE "NULL," AND ONLY THE "FALSE" BRANCH OF THE TESTS NEED BE CONSIDERED. THE CLASS MUST UNDERSTAND THIS TO DO THE EXERCISE AT THE END OF THIS SECTION.

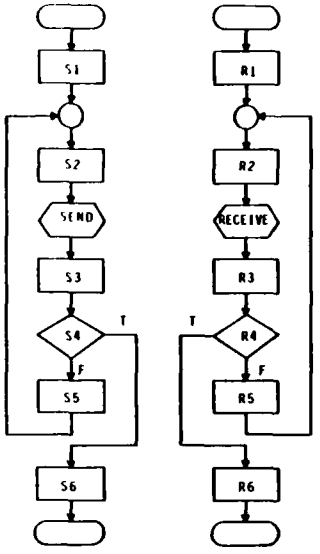
  - ITEM 3: IN THE FLOWCHARTS, LOOP EXIT TESTS OCCUR AFTER THE RENDEZVOUS. (HEXAGONS REPRESENT RENDEZVOUS.)
  - ITEM 4: WHEN WE MERGE TASKS, WE WILL BE ABLE TO OMIT TEST S4 BY TAKING ADVANTAGE OF THE RESULT OF TEST R4. (IT IS ASSUMED THAT THE TESTS HAVE NO SIDE-EFFECTS.)

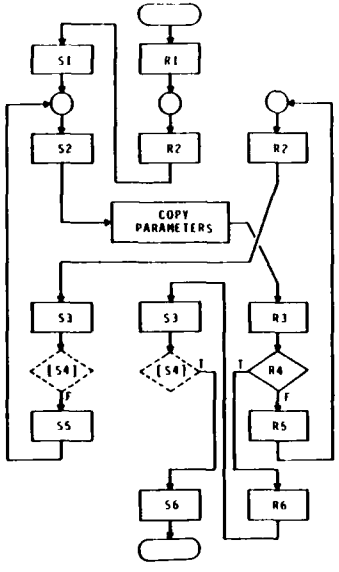
- BULLET 2: CUTTING AND PASTING SIMULATES ONE POSSIBLE SCHEDULING OF THE TWO TASKS, IN WHICH THE RECEIVING TASK STARTS FIRST AND EACH TASK, ONCE EXECUTING, CONTINUES TO EXECUTE UNTIL A RENDEZVOUS IS ENCOUNTERED. THEN THE OTHER TASK TAKES OVER.

  THE RIGHT COLUMN OF THE FLOWCHART CLOSELY CORRESPONDS TO THE ORIGINAL RECEIVING TASK FLOWCHART, BUT ACTIONS BEFORE THE FIRST RECEIVE OPERATION ARE MOVED TO THE MIDDLE COLUMN (WITH R2 DUPLICATED). THE LEFT COLUMN CLOSELY CORRESPONDS TO THE ORIGINAL SENDING TASK FLOWCHART, BUT ACTIONS FOLLOWING THE LAST SEND OPERATION ARE MOVED TO THE MIDDLE COLUMN (WITH S3 AND S4 DUPLICATED). ONE COPY OF S4 IS ENCOUNTERED ONLY AFTER THE CORRESPONDING EVALUATION OF R4 HAS BEEN FOUND TRUE, AND THE OTHER ONLY AFTER IT HAS BEEN FOUND FALSE. THUS THE RESULT OF S4 CAN BE PRESUMED IN EACH CASE, AND THE TEST CAN BE OMITTED. THIS IS INDICATED BY BROKEN-LINE TEST BOXES WITH ONE EXIT EACH.

THE SIMPLE CASE

- CALLING TASK HAS ONE ENTRY CALL STATEMENT, CALLED TASK HAS ONE accept STATEMENT.
  - IF THE ENTRY HAS ONLY in PARAMETERS, THE CALLING TASK IS THE SENDING TASK AND THE CALLED TASK IS THE RECEIVING TASK.
  - IF THE ENTRY HAS ONLY out PARAMETERS, THESE ROLES ARE REVERSED.
  - ASSUME THERE WILL BE AT LEAST ONE RENDEZVOUS. IF ZERO RENDEZVOUS ARE POSSIBLE, THIS CASE CAN BE HANDLED SPECIALLY AFTER TASKS ARE MERGED.
  - IF WE ARE STARTING WITH A CORRECT, DEADLOCK-FREE PROGRAM, THEN TEST S4 AND TEST R4 BECOME TRUE AFTER THE SAME NUMBER OF RENDEZVOUS.





- SOLUTION IS TO CUT AND PASTE THE FLOWCHARTS SO THAT:
  - ACTIONS PRECEDING THE SEND ARE FOLLOWED BY ACTIONS FOLLOWING THE RECEIVE.
  - ACTIONS PRECEDING THE RECEIVE ARE FOLLOWED BY ACTIONS FOLLOWING THE SEND.
  - ACTIONS PRECEDING THE FIRST EXECUTION OF A RECEIVE ARE EXECUTED BEFORE ANYTHING ELSE.
  - ACTIONS FOLLOWING THE LAST EXECUTION OF A RECEIVE ARE EXECUTED AFTER EVERYTHING ELSE.

INSTRUCTOR NOTES

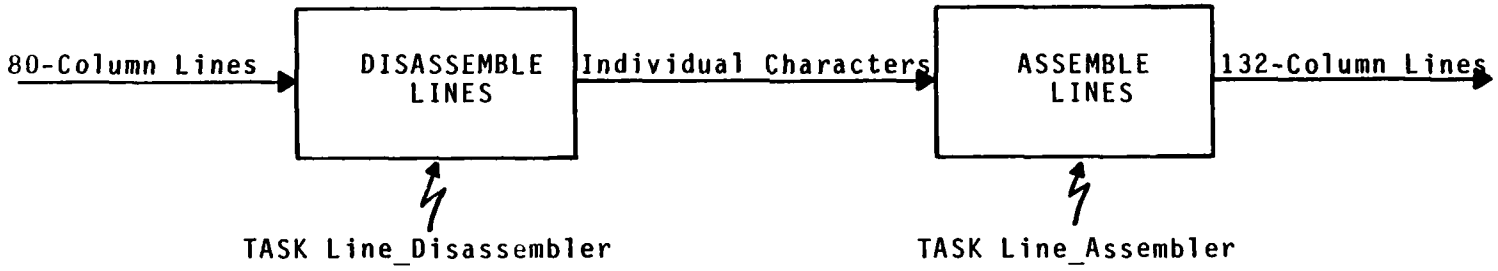
THIS PROBLEM WAS ORIGINALLY INTRODUCED ON SLIDE 15-3.

IT WILL BE USED TO ILLUSTRATE THE MANIPULATION DESCRIBED ON THE PREVIOUS SLIDE.

REFORMATTING REVISITED

- PROBLEM:
  - INPUT FILE OF 80-COLUMN LINES TO BE COPIED, CHARACTER FOR CHARACTER, INTO AN OUTPUT FILE OF 132-COLUMN LINES.
  - BASIC FILE OPERATIONS ARE Read_Line, Write_Line.
  - LAST LINE IN INPUT AND OUTPUT FILES PADDED WITH NULL BYTES.

- STREAM-ORIENTED SOLUTIONS:



- Line_Disassembler AND Line_Assembler CAN BE MERGED.

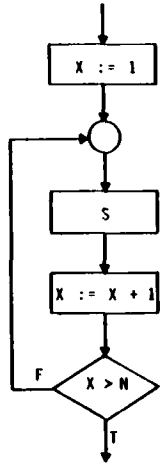
INSTRUCTOR NOTES

THESE FLOWCHARTS ARE BASED ON THE CODE GIVEN ON SLIDE 15-4.

FOR-LOOPS OF THE FORM

```
for X in 1 .. N loop
   S;
end loop;
```

HAVE BEEN DISASSEMBLED INTO INDIVIDUAL STEPS:

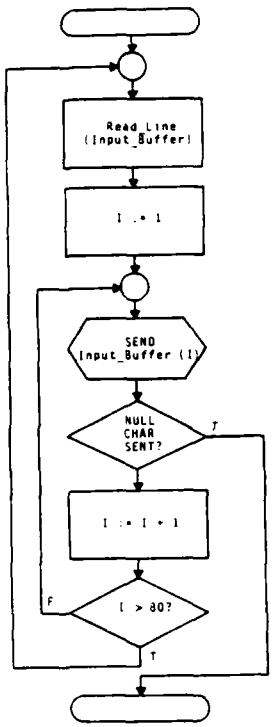


(THIS IS NECESSARY BECAUSE WE WILL BE BRANCHING INTO THE MIDDLE OF WHAT WAS ORIGINALLY A FOR-LOOP.)

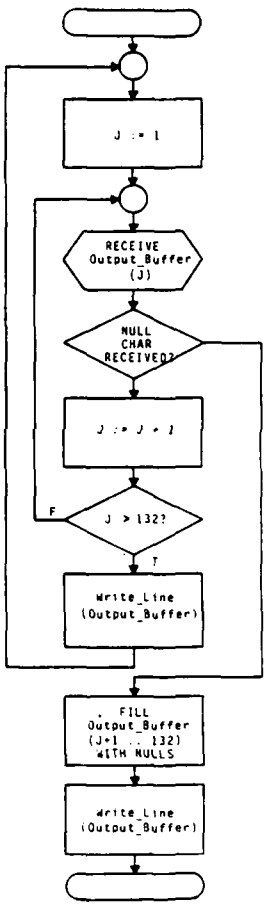
ALSO, BECAUSE WE INTEND TO MERGE THESE FLOWCHARTS, WE MUST RENAME VARIABLES THAT HAVE THE SAME NAME IN BOTH TASKS. IN THIS CASE, THE LOOP VARIABLE I IN Line_Assembler HAS BEEN CHANGED TO J.

Line_Disassembler AND Line_Assembler FLOWCHARTS

Line_Disassembler



Line_Assembler



INSTRUCTOR NOTES

THIS IS THE RESULT OF CUTTING AND PASTING AS DESCRIBED ON SLIDE 24-3.

DON'T GO OVER THE FLOWCHART IN DETAIL. JUST POINT OUT THE FOLLOWING:

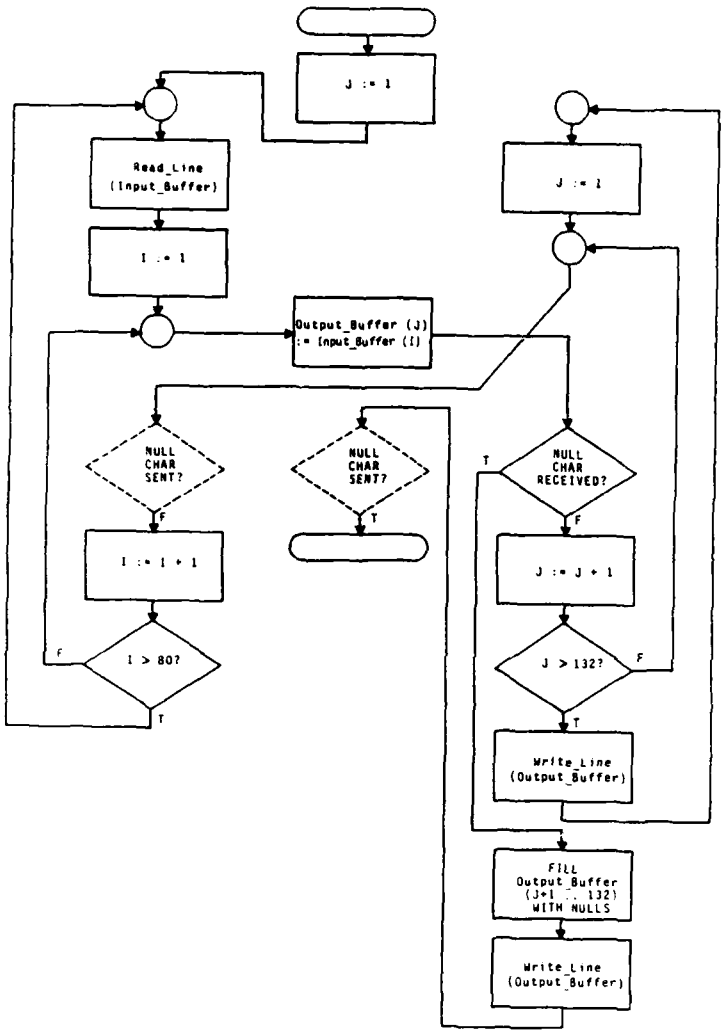
- THE BOX "Output_Buffer (J) := Input_Buffer (I)" CORRESPONDS TO THE "COPY PARAMETERS" BOX ON SLIDE 24-3. Input_Buffer (I) IS THE ENTRY CALL ACTUAL PARAMETER. Output_Buffer (J) IS THE VARIABLE TO WHICH THE accept STATEMENT ASSIGNED THE CORRESPONDING FORMAL PARAMETER. WE HAVE "ELIMINATED THE MIDDLE-MAN" (THE FORMAL PARAMETER).

- THE LOOP-EXIT TEST IN THE ORIGINAL SENDING TASK WAS WHETHER A NULL CHARACTER WAS SENT. THE LOOP-EXIT TEST IN THE ORIGINAL RECEIVING TASK WAS WHETHER A NULL CHARACTER WAS RECEIVED. CLEARLY THESE TESTS ARE REDUNDANT. FOLLOWING SLIDE 24-3, WE ELIMINATE THE REDUNDANT TEST IN THE SENDING TASK.

THE FLOWCHART LOOKS LIKE "SPAGHETTI CODE," BUT CAREFUL EXAMINATION (OR USE OF AN ALGORITHM CALLED INTERVAL ANALYSIS) INDICATES THAT IT ACTUALLY CONSISTS OF TWO NESTED LOOPS, AS SHOWN IN THE CODE ON THE RIGHT. THE INNER LOOP HAS BEEN RE-ASSEMBLED INTO A for-LOOP. (THE UPPER CIRCLE ON THE LEFT IS THE TOP OF Outer_Loop AND THE LOWER CIRCLE ON THE LEFT IS THE TOP OF THE for LOOP. THE if STATEMENT CORRESPONDS TO THE BOTTOM DIAMOND ON THE RIGHT. THE BRANCHES OF THE if STATEMENT REJOIN AT THE LOWER CIRCLE ON THE RIGHT.)

LOOKING AT THIS RE-ASSEMBLED CODE, WE FIND AN EASILY UNDERSTANDABLE ONE-TASK SOLUTION. HOWEVER, THIS SOLUTION WOULD HAVE BEEN DIFFICULT TO DISCOVER WITHOUT FIRST WRITING SEPARATE TASKS AND THEN MERGING THEM (ESPECIALLY IF THE ORIGINAL PROBLEM HAD BEEN MORE COMPLICATED). IT WOULD HAVE BEEN ESPECIALLY DIFFICULT TO HANDLE BOUNDARY CONDITIONS AND PLACEMENT OF TESTS CORRECTLY (E.G. THE CASE WHEN THE LAST OUTPUT LINE CONSISTS OF 132 NULL CHARACTERS).

RESULT OF MERGING




```
J := 1;
Outer_Loop:
   loop
      Read_Line (Input_Buffer);
      for I in 1 .. 80 loop
         Output_Buffer (J) :=
            Input_Buffer (I);
         exit Outer_Loop when
            Output_Buffer (J) = ASCII.NUL;
         J := J + 1;
         if J > 132 then
            Write_Line (Output_Buffer);
            J := 1;
         end if;
      end loop;
   end loop Outer_Loop;
   Output_Buffer (J + 1 .. 132) :=
      (J + 1 .. 132 => ASCII.NUL);
   Write_Line (Output_Buffer);
```

INSTRUCTOR NOTES

WE NOW TURN TO THE SECOND OF THE TWO CASES ON SLIDE 24-2, WHEN THERE ARE MULTIPLE POINTS IN THE TEXT AT WHICH A RENDEZVOUS MAY PLACE.

- BULLET 2: THESE "BUFFER VARIABLES" WILL PLAY A ROLE SIMILAR TO accept STATEMENT FORMAL PARAMETERS.

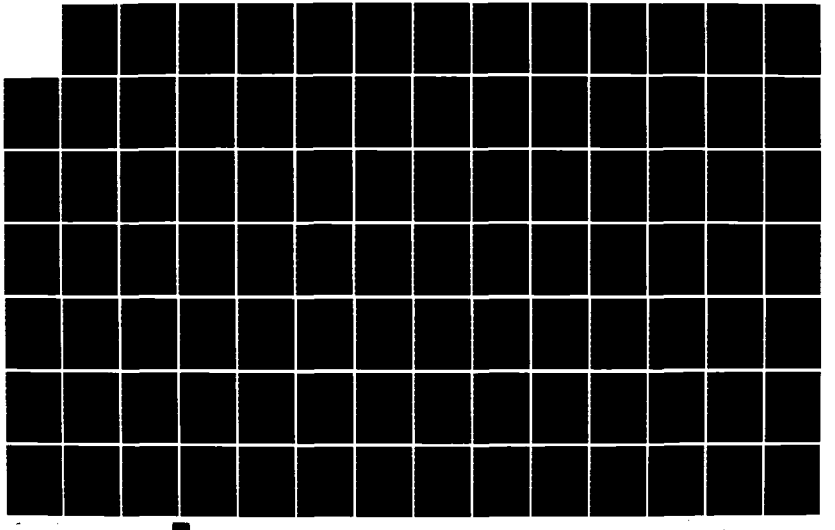
- BULLET 3:
  - ITEM 1: THE RECEIVING TASK FRAGMENT EXECUTED ON THE FIRST ITERATION BRINGS THE RECEIVING TASK UP TO THE POINT WHERE IT IS READY TO RECEIVE ITS FIRST DATA. THEREAFTER, EACH SENDING TASK FRAGMENT PLACES DATA IN BUFFER VARIABLES AND THE RECEIVING TASK FRAGMENT EXECUTED AT THE BEGINNING OF THE NEXT ITERATION EXAMINES THOSE VARIABLES.
  - ITEM 4: THE LAST RECEIVING TASK FRAGMENT EXECUTES WHATEVER ACTIONS FOLLOW THE LAST RECEIVE OPERATION, BRINGING US TO THE END OF THE ORIGINAL RECEIVING TASK'S BODY. THE SECOND case STATEMENT IN THE LOOP MUST THEN BE EXECUTED ONE LAST TIME TO TAKE US FROM THE POINT FOLLOWING THE LAST SEND OPERATION TO THE END OF THE ORIGINAL SENDING TASK'S BODY. THEN BOTH TASKS ARE DONE, SO THE LOOP CAN BE EXITED.

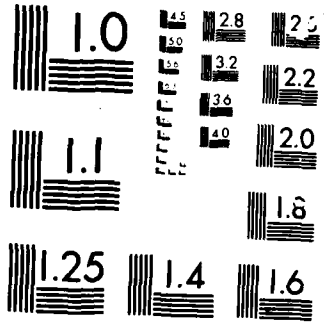
THE COMPLICATED CASE

- CUT EACH FLOW CHART AT RENDEZVOUS POINTS.
  - THIS DIVIDES THE FLOWCHART INTO FRAGMENTS.
  - EACH FRAGMENT BEGINS WITH TASK ACTIVATION OR A RENDEZVOUS.
  - EACH FRAGMENT ENDS WITH TASK COMPLETION OR A RENDEZVOUS.
  - A RENDEZVOUS IN A LOOP MAY BE AT BOTH THE BEGINNING AND END OF ITS FRAGMENT. (THE FRAGMENT WILL RUN FROM THE MIDDLE OF THE LOOP TO THE BOTTOM AND THEN FROM THE TOP OF THE LOOP BACK TO THE MIDDLE.)
- DECLARE BUFFER VARIABLES TO HOLD ENTRY PARAMETERS.
  - SENDING TASK WILL DEPOSIT DATA IN THESE VARIABLES.
  - RECEIVING TASK WILL OBTAIN DATA FROM THESE VARIABLES.
- SIMULATE INTERLEAVING OF FRAGMENTS:
  - A LOOP REPEATEDLY EXECUTES A FRAGMENT OF THE RECEIVING TASK, THEN A FRAGMENT OF THE SENDING TASK.
  - THE LOOP KEEPS TRACK OF WHICH FRAGMENT FROM EACH TASK WILL BE EXECUTED NEXT. IF FRAGMENT a FOR SOME TASK ENDS WITH THE RENDEZVOUS THAT STARTS FRAGMENT b, THEN THE NEXT FRAGMENT FOR THAT TASK IS SET TO b UPON THE COMPLETION OF FRAGMENT a.
  - RESULTING PROGRAM IS A LOOP CONTAINING TWO case STATEMENTS (ONE BASED ON THE NEXT FRAGMENT IN EACH TASK).
  - THE SENDING TASK FRAGMENT THAT ENDS WITH TASK COMPLETION CAUSES THE LOOP TO BE EXITED.

AD-A166 352 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS IN ADA L401 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 5/6

UNCLASSIFIED F/G 5/9 NL





MICROCOPY RESOLUTION TEST CHART

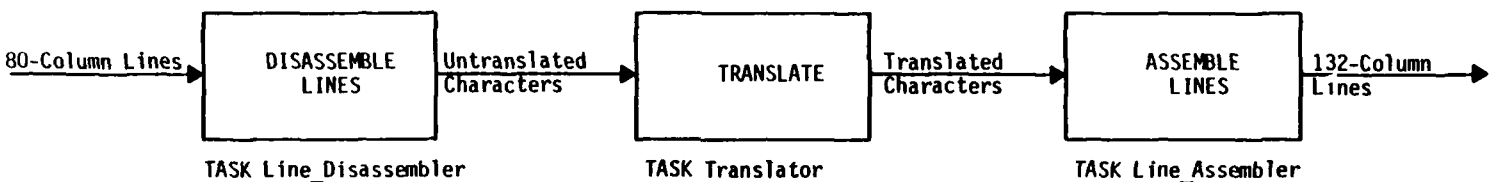
INSTRUCTOR NOTES

THIS MORE COMPLICATED VERSION OF THE REFORMATTING PROBLEM WAS INTRODUCED ON SLIDE 15-14. WE USE IT HERE TO ILLUSTRATE THE MORE COMPLICATED CASE OF MERGING TASKS, SINCE Translator SENDS A CHARACTER TO Line_Assembler FROM FOUR DIFFERENT PLACES.

- BULLET 3:
  - ITEM 3: WE WON'T SHOW THIS HERE.

REVISED REFORMATTING REVISITED

- PROBLEM:
  - BESIDES EXPANDING 80-COLUMN TEXT TO FILL 132-COLUMNS, THE REFORMATTER SHOULD TRANSLATE EACH OCCURRENCE OF "**" IN THE INPUT TO " " IN THE OUTPUT.
  - THE TRANSLATION SHOULD TAKE PLACE EVEN IF THE "**" IS SPLIT ACROSS LINES.
- STREAM-ORIENTED SOLUTION:



- Translator AND Line_Assembler CAN BE MERGED.
  - Translator HAS THREE DIFFERENT ENTRY CALL STATEMENTS CALLING Line_Assembler.Deliver_Character, SO THIS IS THE MORE COMPLICATED CASE.
  - RENDEZVOUS WITH Line_Disassembler WILL BE TREATED AS ORDINARY OPERATIONS (RECTANGLES IN THE FLOWCHART).
  - AFTER MERGING THESE TASKS, WE COULD, IN PRINCIPLE, MERGE THE RESULT WITH Line_Disassembler.

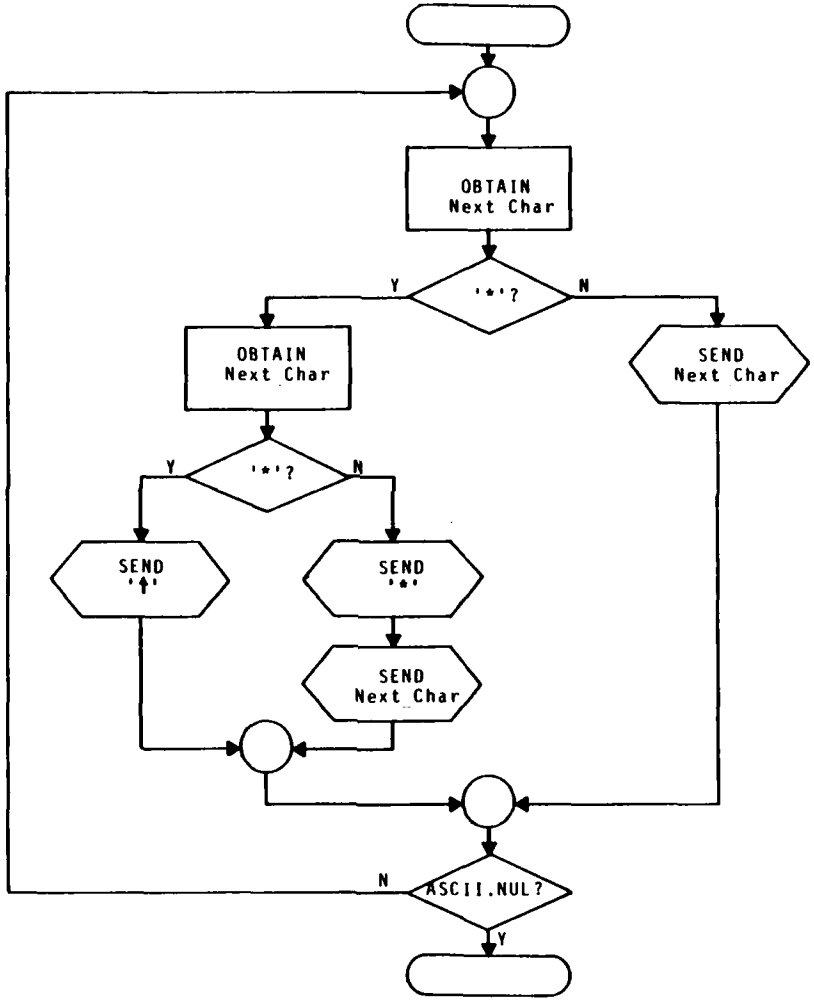
INSTRUCTOR NOTES

THIS FLOWCHART IS DERIVED FROM THE CODE ON SLIDE 15-18.

THE "OBTAIN Next_Char" OPERATION IS REALLY AN accept STATEMENT FOR AN ENTRY CALL MADE BY Line_Disassembler, BUT WE TREAT IT AS A PRIMITIVE OPERATION WHEN MERGING Translator AND Line_Assembler. WE THUS REPRESENT IT WITH A RECTANGLE RATHER THAN A HEXAGON.

TRACE THROUGH THE FLOWCHART QUICKLY. THEN POINT OUT THE FOUR DIFFERENT PLACES AT WHICH SEND OPERATIONS OCCUR.

FLOWCHART FOR TRANSLATOR



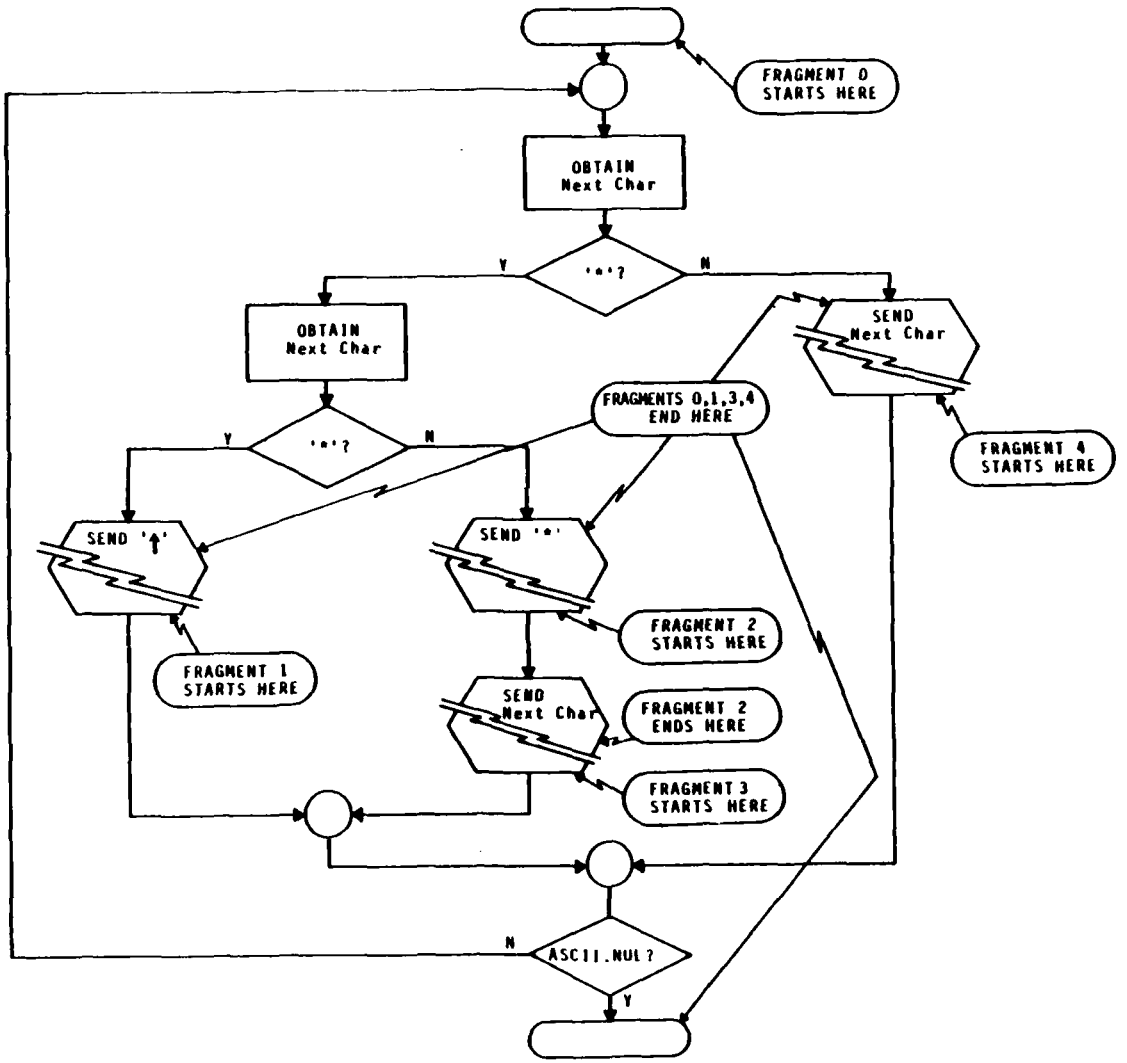
INSTRUCTOR NOTES

TRACE THROUGH EACH FRAGMENT. EACH ONE STARTS AT ONE  OR  SYMBOL AND ENDS AT ONE OR MORE SUCH SYMBOLS (ONE OF WHICH MIGHT BE THE SAME  WITH WHICH IT STARTED). CERTAIN BOXES OCCUR ON MORE THAN ONE FRAGMENT.

NUMBERING OF FRAGMENTS IS ARBITRARY, THOUGH IT IS A GOOD CONVENTION TO USE ZERO FOR THE STARTING FRAGMENT.

FRAGMENTS 1, 3, AND 4 ARE REALLY IDENTICAL, BECAUSE EACH LEADS DIRECTLY TO THE BOTTOMMOST DIAMOND BEFORE PERFORMING ANY ACTION. WE SHALL TAKE ADVANTAGE OF THIS OBSERVATION WHEN WRITING A LOOP TO INTERLEAVE FRAGMENTS OF Translator AND Line_Assembler.

DIVISION INTO FRAGMENTS

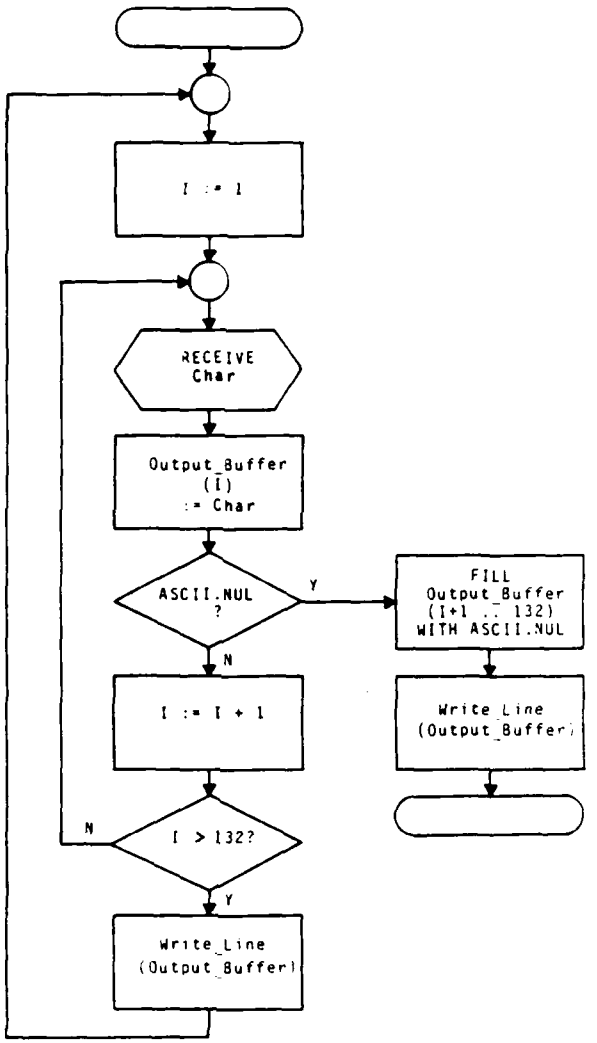


INSTRUCTOR NOTES

THIS IS ESSENTIALLY THE FLOWCHART FOR Line_Assembler GIVEN ON SLIDE 24-5, BUT WE DO NOT RENAME I, AND THE ASSIGNMENT TO Output_Buffer (I) IS MADE EXPLICIT. THE VARIABLE Char IS THE BUFFER VARIABLE MENTIONED ON SLIDE 24-7.

THIS FLOWCHART IS GIVEN AGAIN TO REFRESH STUDENTS' MEMORIES. MOVE ON QUICKLY.

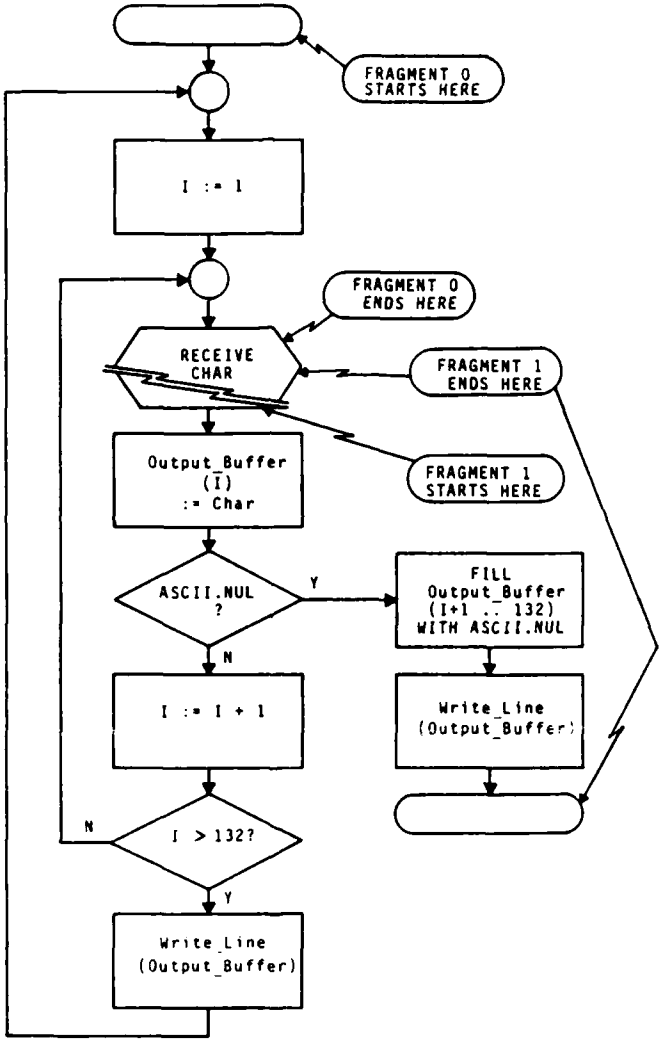
FLOWCHART FOR Line_Assembler



INSTRUCTOR NOTES

THIS SLIDE IS PRESENTED FOR COMPLETENESS, BUT ILLUSTRATES THE SAME POINTS AS SLIDE 24-10. MOVE ON QUICKLY.

DIVSION INTO FRAGMENTS



INSTRUCTOR NOTES

SLIDES 24-13 AND 24-14 ILLUSTRATE THE METHOD DESCRIBED ON SLIDE 24-7.

FOR EACH ORIGINAL TASK, WE DECLARE AN ENUMERATION TYPE FOR NAMING THAT TASK'S FRAGMENTS. WE TREAT FRAGMENTS 1, 3, AND 4 OF SLIDE 24-10 (FROM THE Translator TASK) AS A SINGLE FRAGMENT.

WE DECLARE A Next-Fragment VARIABLE FOR EACH TASK, INITIALIZED TO THE TASK'S INITIAL FRAGMENT.

BEFORE REVIEWING STATEMENTS ON SLIDE 24-13, QUICKLY SHOW THE CLASS SLIDE 24-14 SO THAT THEY SEE THE GENERAL STRUCTURE OF A LOOP CONTAINING A CASE STATEMENT FOR Next_Assembler_Fragment FOLLOWED BY A CASE STATEMENT FOR Next_Translator_Fragment. Line_Assembler IS THE RECEIVING TASK AND Translator IS THE SENDING TASK.

IF TWO PROJECTORS ARE AVAILABLE, PLACE SLIDE 24-12 ON THE OTHER PROJECTOR WHEN GOING OVER THE CASE STATEMENT ON SLIDE 24-13. EACH ARM OF THE CASE STATEMENT CORRESPONDS TO A FRAGMENT ON SLIDE 24-12. GENERALLY, THE LAST THING A FRAGMENT DOES IS SET Next_Assembler_Fragment TO THE NAME OF THE FRAGMENT THAT FOLLOWS NEXT ON THE FLOWCHART. THE EXCEPTION IS AT THE END OF THE then PART IN THE A_1 CASE. THIS CORRESPONDS TO REACHING THE END OF THE Line_Assembler TASK BODY. THIS ONLY HAPPENS ON THE LAST ITERATION OF THE MERGED LOOP, SO THERE IS NO NEED TO SET A NEXT FRAGMENT.

(THERE IS AN OBVIOUS SIMPLIFICATION POSSIBLE BECAUSE THE NEXT Line_Assembler FRAGMENT IS ALWAYS SET TO A_1. DEFER DISCUSSION OF THIS TO SLIDE 24-15.

BODY OF MERGED TASK (1 OF 2)

```
task body Translator_And_Assembler is
   type Translator_Fragment_Type is (T_0, T_1_3_4, T_2);
   type Assembler_Fragment_Type is (A_0, A_1);
   Next_Translator_Fragment : Translator_Fragment_Type := T_0;
   Next_Assembler_Fragment  : Assembler_Fragment_Type := A_0;
   Input_Buffer     : String (1 .. 80);
   Output_Buffer    : String (1 .. 132);
   Next_Char, Char : Character;
   I                : Positive;
begin
   loop
      case Next_Assembler_Fragment is
         when A_0 =>
            I := 1;
            Next_Assembler_Fragment := A_1;
         when A_1 =>
            Output_Buffer (I) := Char;
            if Char = ASCII.NUL then
               Output_Buffer (I+1 .. 132) := (I+1 .. 132 =>  ASCII.NUL);
               Write_Line (Output_Buffer);
            else
               I := I + 1;
               if I > 132 then
                  Write_Line (Output_Buffer);
                  I := 1;
               end if;
               Next_Assembler_Fragment := A_1;
            end if;
      end case;
```

INSTRUCTOR NOTES

IF TWO PROJECTORS ARE AVAILABLE, PLACE SLIDE 24-10 ON THE OTHER PROJECTOR WHEN GOING OVER THE CASE STATEMENT ON SLIDE 24-14.

AGAIN, RELATE EACH ARM OF THE CASE STATEMENT TO A FRAGMENT OF THE FLOWCHART ON SLIDE 24-10. SHOW HOW THE LAST ACTION IN EACH CASE IS TO SET Next_Translator_Fragment ACCORDING TO THE NEXT FRAGMENT ON THE FLOWCHART.

BODY OF MERGED TASK (2 OF 2)

```
      case Next_Translator_Fragment is
         when T_0 =>
            accept Deliver_Char (c : in Character) do
               Next_Char := c;
            end Deliver_Char;
            if Next_Char = '*' then
               accept Deliver_Char (c : in Character) do
                  Next_Char := c;
               end Deliver_Char;
               if Next_Char = '*' then
                  Char := '↑';
                  Next_Translator_Fragment := T_1_3_4;
               else
                  Char := '*';
                  Next_Translator_Fragment := T_2;
               end if;
            else
               Char := Next_Char;
               Next_Translator_Fragment := T_1_3_4;
            end if;
         when T_1_3_4 =>
            exit when Next_Char = ASCII.NUL;          --≫ LOOP EXIT ≪--
            [copy of the statements for the case above]
         when T_2 =>
            Char := Next_Char;
            Next_Translator_Fragment := T_1_3_4;
      end case;

   end loop;

end Translator_And_Assembler;
```

INSTRUCTOR NOTES

- BULLET 1: THE MERGED PROGRAM IS DERIVED MECHANICALLY AND THUS DOES NOT TAKE ADVANTAGE OF CERTAIN OBVIOUS OPPORTUNITIES FOR IMPROVEMENT.

  FOR EXAMPLE, THE A_0 AND T_0 CASES ALWAYS OCCUR ON THE FIRST ITERATION OF THE LOOP, AND ONLY THEN. THE ACTIONS PERFORMED ON THE FIRST ITERATION CAN BE MOVED OUT OF THE LOOP AND EXECUTED BEFORE THE LOOP IS ENTERED. THESE ACTIONS CONSIST OF THE A_0 ARM OF THE FIRST CASE STATEMENT FOLLOWED BY THE T_0 ARM OF THE SECOND CASE STATEMENT. THE ENUMERATION LITERALS A_0 AND T_0 CAN THEN BE REMOVED FROM THE PROGRAM AND THE CORRESPONDING ARMS CAN BE REMOVED FROM THE CASE STATEMENTS. THIS LEAVES Assembler_Fragment_Type WITH ONE ENUMERATION VALUE (A_1) AND THE TOP case STATEMENT WITH ONE ARM. THUS Assembler_Fragment_Type AND Next_Assembler_Fragment CAN BE REMOVED, AND THE STATEMENTS IN THE A_1 ARM CAN BE EXECUTED UNCONDITIONALLY EACH TIME THROUGH THE LOOP.

  MERGING TASKS MAY ALSO ALLOW THE COMPILER TO PERFORM CERTAIN OPTIMIZATIONS THAT WOULD HAVE BEEN TOO DIFFICULT TO FIND WHEN OPERATIONS WERE SPLIT ACROSS TWO TASKS.

- BULLET 2: PRESUMABLY, THIS PRAGMA WOULD LEAD THE COMPILER TO DISCOVER THE ADDITIONAL OPTIMIZATIONS MENTIONED ABOVE.

  IN THE ABSENCE OF SUCH A PRAGMA, PROGRAMMERS SHOULD RETAIN COPIES OF THE ORIGINAL, UNMERGED TASKS AS A FORM OF DOCUMENTATION.

CONCLUSIONS

- MERGED PROGRAMS CAN USUALLY BE FURTHER SIMPLIFIED AND TUNED.

- IDEALLY, WE WOULD PREFER AN IMPLEMENTATION-DEFINED PRAGMA, SAY

  pragma Merge (Translator, Line_Assembler);

  THAT WOULD CAUSE THE COMPILER TO PERFORM THESE MANIPULATIONS AUTOMATICALLY WHEN GENERATING OBJECT CODE.

  THE SOURCE FILE COULD THEN REMAIN INTACT IN ITS ORIGINAL, SIMPLE FORM.

- THE ABILITY TO MERGE TASKS MAKES STREAM-ORIENTED TASK DESIGN FEASIBLE.
  - FIRST, WRITE A CLEAR, SIMPLE PROGRAM, POSSIBLE CONTAINING MANY TASKS.
  - SECOND, SEE WHETHER THE NUMBER OF TASKS MAKES THE PROGRAM UNACCEPTABLY EXPENSIVE.
  - THIRD, TRANSFORM THE SIMPLE BUT INEFFICIENT SOLUTION INTO AN OBSCURE BUT EFFICIENT ONE IF NECESSARY.

INSTRUCTOR NOTES

THE FOLLOWING TASK BODIES ARE FROM THE SOLUTION TO EXERCISE 15.1.

```
task body Packet_Disassembler is
   Packet : Packet_Type;
begin
   loop
      Multiplexer_Task.Get_Packet (Packet);
      for I in 1 .. 3 loop
         Averager.Deliver_Reading (Packet (I));
      end loop;
   end loop;
end Packet_Disassembler;

task body Averager is
   Next_Reading, Sum : Float;
begin
   loop
      Sum := 0;
      for I in 1 .. 5 loop
         accept Deliver_Reading (Reading : in Float) do
            Next_Reading := Reading;
         end Deliver_Reading;
         Sum := Sum + Next_Reading;
      end loop;
      Position_Calculator.Deliver_Average (Sum/5.0);
   end loop;
end Averager;
```

DRAW FLOWCHARTS FOR THESE TASKS AND THEN DRAW A FLOWCHART FOR THE RESULT OF MERGING THESE TASKS.

INSTRUCTOR NOTES

- ALLOW 120 MINUTES FOR THIS SECTION.

- GO THROUGH THE EXAMPLES CAREFULLY, MAKING SURE THE CLASS UNDERSTANDS WHERE THE PERFORMANCE PROBLEMS ARE AND HOW THEY ARE SOLVED.

Section 25

NON-CONCURRENT TUNING

INSTRUCTOR NOTES

- WE JUSTIFY THE TUNING WE PERFORM ON THE TWO ASSUMPTIONS. WE DO NOT PERFORM THE TUNING ON LARGE PARTS OF THE PROGRAM, ONLY ON THOSE PARTS THAT HAVE PERFORMANCE PROBLEMS.

TUNING

- TUNING IS IMPORTANT IN ANY TIME-CRITICAL PROGRAM REGARDLESS OF WHETHER CONCURRENCY IS BEING USED.

- IN THIS SECTION SEVERAL TUNING METHODS ARE PRESENTED.

  - ALGORITHM IMPROVEMENT
  - FIXED POINT ARITHMETIC
  - SUBPROGRAM CALL IMPROVEMENT
  - REMOVING RECURSION
  - LOOP UNFOLDING
  - SUPPRESSING RUNTIME CHECKS
  - SELECTIVE RECODING IN ASSEMBLY LANGUAGE.

- THE FOLLOWING ASSUMPTIONS HOLD THROUGHOUT THIS SECTION.

  - PERFORMANCE PROBLEMS HAVE BEEN DETECTED IN THE PROGRAM PIECES WE LOOK AT. THE PROGRAM IS UNACCEPTABLE UNLESS "SUBSTANTIAL" IMPROVEMENT IS ACHIEVED IN THE PROGRAM PIECES.

INSTRUCTOR NOTES

- CODE SIMPLIFICATION

  - SIMPLE PROGRAMS CAN BE TUNED; COMPLEX PROGRAMS CANNOT BE. PIECES OF CODE THAT ARE COMPLEX MAY NEED TO BE REWRITTEN TO MAKE THEM UNDERSTANDABLE. IF YOU CAN'T UNDERSTAND IT, YOU CAN'T TUNE IT. OF COURSE YOU MUST EVENTUALLY BE ABLE TO UNDERSTAND A COMPLEX PIECE OF CODE IN ORDER TO REWRITE IT, BUT FINDING TUNING OPPORTUNITIES IN COMPLEX CODE IS DIFFICULT. IN ADDITION, ANY TUNING WILL JUST MAKE THE CODE EVEN MORE COMPLEX.

- PROBLEM SIMPLIFICATION

  - BREAKING THE CODE INTO SMALLER SIMPLER PROBLEMS CAN RESULT IN PIECES THAT CAN BE HANDLED SEPARATELY. FOR EXAMPLE, A PIECE THAT OCCURS FREQUENTLY AND CAN BE HIGHLY TUNED, AND A PIECE THAT OCCURS MUCH LESS FREQUENTLY BUT IS NOT EFFICIENT.

- EARLY BINDING

  - AN EXAMPLE IS PRECOMPUTING FUNCTION RESULTS, E.G. FIBONACCI NUMBERS OR THE CODEWORD COUNTING EXAMPLE FROM PART 22.

BENTLEY'S FUNDAMENTAL RULES

- FOUR FUNDAMENTAL RULES IN APPLYING THE TECHNIQUES OF THIS SECTION.
  - CODE SIMPLIFICATION: MOST FAST PROGRAMS ARE SIMPLE THEREFORE, KEEP CODE SIMPLE TO MAKE IT FASTER.
  - PROBLEM SIMPLIFICATION: TO INCREASE THE EFFICIENCY OF A PROGRAM, SIMPLIFY THE PROBLEM IT SOLVES.
  - RELENTLESS SUSPICION: QUESTION THE NECESSITY OF EACH INSTRUCTION IN A TIME-CRITICAL PIECE OF CODE.
  - EARLY BINDING: MOVE WORK FORWARD IN TIME. SPECIFICALLY, DO WORK NOW JUST ONCE IN HOPE OF AVOIDING DOING IT MANY TIMES LATER.
- MANY OF THE TECHNIQUES AND EXAMPLES IN THIS SECTION ARE DERIVED FROM BENTLEY'S, WRITING EFFICIENT PROGRAMS. THIS BOOK DESCRIBES MANY MORE TECHNIQUES THAN CAN BE DESCRIBED IN THIS SECTION.

INSTRUCTOR NOTES

- BULLET 1

  MANY TIMES A SIMPLE ALGORITHM IS USED BECAUSE

  - NEED FOR MORE SOPHISTICATED ONE NOT APPARENT.
  - PROGRAMMER MAY NOT KNOW MORE SOPHISTICATED ALGORITHM.
  - STILL RESEARCHING MORE SOPHISTICATED ONE
    - SYSTEM MAY NEED TO BE COMPLETED WITH LESS SOPHISTICATED ALGORITHM TO LET INTEGRATION PROBLEMS BE UNCOVERED, GIVE DEMOS, ETC.

- BULLET 2

  - REMEMBER THAT TUNING OFTEN ADDS COMPLEXITY. WE WILL SEE EXAMPLES LATER IN THE SECTION. A NEW ALGORITHM GENERALLY WILL NOT ADD AS MUCH COMPLEXITY AS SOME OF THE TECHNIQUES PRESENTED LATER IN THIS SECTION.

  - MANY TUNING TECHNIQUES MOVE STATEMENTS, E.G. OUT OF LOOPS, ETC. SUCH MOVEMENT COULD OBSCURE THE BOUNDARY BETWEEN THE ALGORITHM AND THE TEXT THAT SURROUNDS IT.

- BULLET 3

  - SEARCHING IS USED SINCE IT IS ONE OF THE BEST STUDIED TOPICS WITH MUCH INFORMATION ON ALGORITHMS AND TIMINGS. ALSO, THE SEARCHING ALGORITHMS WE USE ARE EASY TO UNDERSTAND AND CAN BE USED TO ILLUSTRATE SOME LATER TUNING TECHNIQUES.

IMPROVING THE ALGORITHM

- SUBSTANTIAL PERFORMANCE IMPROVEMENTS CAN OFTEN BE ACHIEVED BY IMPROVING THE ALGORITHMS BEING USED.

- TYPICALLY, THIS SHOULD BE THE FIRST AREA TO INVESTIGATE FOR TUNING.

  - BEST POTENTIAL FOR PERFORMANCE IMPROVEMENT.
  - LEAST DESTRUCTION OF PROGRAM CLARITY.
  - OTHER TUNING TECHNIQUES MAY "SPREAD" THE ALGORITHM THROUGH THE PROGRAM.

- SEARCHING WILL BE USED TO ILLUSTRATE THE POTENTIAL FOR IMPROVEMENT.

INSTRUCTOR NOTES

- THIS ROUTINE ASSUMES THAT Table ALWAYS HAS A LOWER INDEX OF 1, AS DO THE OTHER SEARCHING ALGORITHMS.
- MANY OF THE CODE PIECES IN THIS SECTION ARE DERIVED FROM THE PASCAL VERSIONS IN BENTLEY'S BOOK. THE TIMINGS USED ARE BASED ON HIS MEASUREMENTS. THIS GIVES SOME CONCRETENESS TO PERFORMANCE IMPROVEMENT. DO NOT MENTION THIS TO THE CLASS.
- LINEAR SEARCHING WAS PRESENTED IN L305, BUT REVIEW IT ANYWAY. DO NOT SPEND MUCH TIME. WE JUST WANT THEM TO UNDERSTAND WHY IT IS INEFFICIENT.
- THE LINEAR SEARCH STARTS AT THE BEGINNING OF THE ARRAY AND SEARCHES FORWARD UNTIL IT FINDS THE ITEM BEING SEARCHED FOR OR REACHES THE END OF THE ARRAY. AVERAGE PERFORMANCE IS ORDER (n/2) WITH WORST CASE ORDER (n), WHERE n IS THE TABLE SIZE. FOR THE LINEAR SEARCH SHOWN, ORDER (n/2) = 3.65 n.
- FOR EACH OF THE ALGORITHMS WE LOOK AT ONE MILLION SEARCHES OF AN ARRAY WITH 1,000 ELEMENT AND 10,000 ELEMENTS.
- THE POOR PERFORMANCE OF THE LINEAR SEARCH IS DOMINATED BY ITS BRUTE FORCE APPROACH OF CHECKING EACH ELEMENT. HOWEVER, THERE IS A SUBTLE INEFFICIENCY THAT ACCOUNTS FOR ALMOST HALF THE TIME. FOR EACH ELEMENT, TWO CONDITIONS ARE TESTED: HAS THE END OF THE TABLE BEEN REACHED AND THEN IF NOT, THEN IS THIS THE ELEMENT. THE NEXT SLIDE SHOWS HOW THE TEST IS MADE AN IMPLICIT PART OF THE SECOND TEST BY USING A SENTINEL.

LINEAR SEARCH

```
procedure Linear_Search (Item  : in Float;
                         Table : in Table_Type;
                         Index : out Natural) is

   I : Natural;

begin
   I : = 1;
   while I <= Table'Last and then Table (I) /= Item loop
       I := I + 1;
   end loop;
   if I <= Table'Last then
       Index := I;                    -- Found
   else
       Index := 0;                    -- Not Found
   end if;
end Linear_Search;
```

- PERFORMANCE
  - AVERAGE CASE = 3.65n MICROSECONDS/SEARCH
  - 1,000,000 SEARCHES
    - 1,000 ELEMENT TABLE
      - 3.65 MILLISECONDS/SEARCH
      - TIME FOR 1,000,000 SEARCHES = 3650 SECONDS ≈ 61 MINUTES
    - 10,000 ELEMENT TABLE
      - 36.5 MILLISECONDS/SEARCH
      - TIME FOR 1,000,000 SEARCHES = 36,500 SECONDS ≈ 10 HOURS.

INSTRUCTOR NOTES

- THE LINEAR SEARCH IS IMPROVED BY USING THE LAST ELEMENT OF THE ARRAY AS A SENTINEL. THE ITEM BEING SEARCHED FOR IS PLACED IN THE LAST ARRAY POSITION. THE LINEAR SEARCH WILL EVENTUALLY FIND THE ELEMENT. IF IT FINDS IT AT THE SENTINEL POSITION, THEN THE ELEMENT WAS NOT IN THE ORIGINAL ARRAY - THIS CORRESPONDS TO THE I < Table'Last TEST IN THE While LOOP OF THE PREVIOUS VERSION.

- NOTE THAT THE TABLE SIZE IS ASSUMED TO BE 1 LARGER THAN NEEDED TO ALLOW FOR THE SENTINEL. THIS IS NOT A MUST: WE COULD DECLARE AN OBJECT Last_Element : Float. THEN AT THE START OF THE PROCEDURE WE CAN REPLACE.

```
Table (Table'Last) := Item;
```

WITH

```
Last_Element := Table (Table'Last);
Table (Table'Last) := Item;
```

AND THE IF STATEMENT COULD BE REPLACED WITH

```
if I < Table'Last then
   Index := I;
elsif Item = Last_Element then
   Index := Table'Last;
else
   Index := 0;
end if;
Table (Table'Last) := Last_Element;
```

- NOTE THAT Table MUST NOW BE in out INSTEAD OF in.

- THE VERTICAL BARS ON THE LEFT MARK LINES THAT HAVE CHANGED.

LINEAR SEARCH WITH SENTINEL

```
procedure Linear_Search_With_Sentinel (Item  : in Float;
                                       Table : in out Table_Type;
                                       Index : out Natural) is

   I : Natural;

begin
   Table (Table'Last) := Item;
   I := 1;
   while Table (I) /= Item loop      -- "I  <= Table'Last" removed
      I := I + 1;
   end loop;
   if I < Table'Last then
      Index := I;                    -- Found
   else
      Index := 0;                    -- Not Found
   end if;
end Linear_Search_With_Sentinel;
```

- PERFORMANCE
  - AVERAGE CASE = 2.05n MICROSECONDS/SEARCH
  - 1,000,000 SEARCHES
    - 1,000 ELEMENT TABLE
      - 2.05 MILLISECONDS/SEARCH
      - TIME FOR 1,000,000 SEARCHES = 2050 SECONDS ≈ 34 MINUTES
    - 10,000 ELEMENT TABLE
      - 20.5 MILLISECONDS/SEARCH
      - TIME FOR 1,000,000 SEARCHES = 20,500 SECONDS ≈ 6 HOURS.

INSTRUCTOR NOTES

- BINARY SEARCHES WERE DISCUSSED IN L305. JUST DISCUSS IT ENOUGH TO LET THE CLASS UNDERSTAND WHY THIS IS AN IMPROVEMENT.

- THE BINARY SEARCH WORKS BY REPEATEDLY DIVIDING THE Table'Range IN HALF. IF THE ELEMENT AT THE Mid_Point IS THE ITEM BEING SEARCHED FOR THEN THE SEARCH IS SUCCESSFUL. OTHERWISE, SINCE THE Table IS SORTED, IF THE ITEM IS IN THE Table THEN IT MUST BE IN Table (1 .. Mid_Point -1) IF THE ITEM IS LESS THAN THE Mid_Point ELEMENT, OR IN Table (Mid_Point + 1 .. Table'Last) IF THE ITEM IS GREATER THAN THE Mid_Point ELEMENT, THUS THE SEARCH IS REPEATED ON THE APPROPRIATE SLICE. THIS CONTINUES UNTIL THE ITEM IS FOUND OR THE SELECTED SLICE IS NULL. IN THE LATTER CASE, THIS MEANS THE ELEMENT IS NOT IN THE ORIGINAL TABLE.

- AT EACH REPETITION, THE BINARY SEARCH RULES OUT HALF OF THE REMAINING ELEMENTS. THUS THE BINARY SEARCH SPENDS LESS TIME LOOKING AT ELEMENTS THAN THE LINEAR SEARCHES. IN FACT ITS AVERAGE AND WORST CASE ARE IDENTICAL AT ORDER ($\log_2 n$). IN PARTICULAR, IN A 1,000 ELEMENT TABLE IT LOOKS A $\log_2 1000 \approx 10$ ELEMENTS WHILE IN A 10,000 ELEMENT TABLE IT LOOKS AT $\log_2 10{,}000 \approx 13$ ELEMENTS.

```
procedure Binary_Search (Item  : in Float;
                         Table : in Table_Type;
                         Index : out Natural) is
   Mid_Point : Natural;
   Lower     : Natural := Table'First;
   Upper     : Natural := Table'Last;
begin
   if Lower > Upper then
      Index := 0;
   else
      Mid_Point := (Lower + Upper) /2;
      if Item = Table (Mid_Point) then
         Index := Mid_Point;
      elsif Item < Table (Mid_Point) then
         Binary_Search (Item, Table (Lower .. Mid_Point -1), Index));
      else
         Binary_Search (Item, Table (Mid_Point +1, Upper), Index);
      end if;
   end if;
end Binary_Search;
```

- PERFORMANCE
  - WORST CASE = AVERAGE CASE = 23 $\log_2$ n MICROSECONDS/SEARCH
  - 1,000,000 SEARCHES
    - 1,000 ELEMENT TABLE
      - 23 $\log_2$ 1000 MICROSECONDS/SEARCH $\approx$ 230 MICROSECONDS/SEARCH
      - TIME FOR 1,000,000 SEARCHES = 230 SECONDS $\approx$ 3.8 MINUTES
    - 10,000 ELEMENT TABLE
      - 23 $\log_2$ 10,000 MICROSECONDS/SEARCH $\approx$ 299 MICROSECONDS/SEARCH
      - TIME FOR 1,000,000 searches = 299 SECONDS $\approx$ 5 MINUTES.

INSTRUCTOR NOTES

- THIS SLIDE SUMMARIZES THE PERFORMANCE RESULTS OF THE PREVIOUS SLIDES.

COMPARING RESULTS

Algorithm	Microseconds/ Search	One Million Searches For Table Size n	
		n = 1000	n = 10,000
Linear Search	3.65 n	60.8 MINUTES	10.1 HOURS
Linear Search with Sentinel	2.05 n	34.1 MINUTES	6.3 HOURS
Binary Search	23 $\log_2$ n	3.8 MINUTES *	5.0 MINUTES **

* 16 TIMES FASTER THAN LINEAR SEARCH WITHOUT SENTINEL

** 122 TIMES FASTER THAN LINEAR SEARCH WITHOUT SENTINEL

INSTRUCTOR NOTES

- THIS BINARY SEARCH IS INCLUDED FOR COMPLETENESS. IT WILL ALSO BE USED AS AN EXAMPLE FOR SELECTIVE ASSEMBLY LANGUAGE CODING.

- IN THE CASE WHEN THE TABLE SIZE IS FIXED, THIS ALGORITHM FOR A BINARY SEARCH CAN BE USED. OF COURSE THERE IS A CORRESPONDING LACK OF GENERALITY AND INCREASE IN COMPLEXITY, BUT REMEMBER THAT THE PREMISE OF THIS SECTION IS THAT PERFORMANCE MUST BE DRASTICALLY IMPROVED. CLARITY CAN BE SACRIFICED FOR PERFORMANCE IN A FEW CRITICAL CODE SECTIONS, BUT SUCH SACRIFICES SHOULD BE MINIMIZED.

KNUTH'S BINARY SEARCH

```
procedure Binary_Search (Item : in Float;
                         Table : in Table_Type;
                         Index : out Natural) is
   Lower : Natural := 1;
begin
   if Table (512) < Item then
       Lower := 1000 + 1 - 512;
   end if;
   if Table (Lower + 256) < Item then
       Lower := Lower + 256;
   end if;
   if Table (Lower + 128) < Item then
       Lower := Lower + 128;
   end if;
   if Table (Lower + 64) < Item then
       Lower := Lower + 64;
   end if;
   if Table (Lower + 32) < Item then
       Lower := Lower + 32;
   end if;
   if Table (Lower + 16) < Item then
       Lower := + 16;
   end if;
   if Table (Lower + 8) < Item then
       Lower := Lower + 8;
   end if;
   if Table (Lower + 4) < Item then
       Lower := Lower + 4;
   end if;
   if Table (Lower + 2) < Item then
       Lower := Lower + 2;
   end if;
   if Table (lower + 1) < Item then
       Lower := Lower + 1;
   end if;
   if Lower <= 1000 and then Table (Lower) = Item then
       Index := Lower;
   else
       Index := 0;
   end if;
end Binary_Search;
```

- EXECUTION TIME : 5 $\log_2$ n
  - 1,000,000 SEARCHES
    - 1000 ELEMENT TABLE
      - 50 SECONDS
      - 73 TIMES FASTER THAN LINEAR SEARCH WITHOUT SENTINEL
    - 10,000 ELEMENT TABLE
      - 1.1 MINUTES
      - 553 TIMES FASTER THAN LINEAR SEARCH WITHOUT SENTINEL
- CLARITY SACRIFICED FOR PERFORMANCE

INSTRUCTOR NOTES

- BULLET 1

  ONCE WE DISCOVER A BETTER ALGORITHM, WE CAN SOMETIMES EXPLOIT SPECIFIC KNOWLEDGE - SUCH AS THE ONLY TABLE SEARCHED HAS 1000 ELEMENTS - TO USE SPECIAL CASES OF THE ALGORITHM.

- BULLET 2

  THE EXAMPLE IN BULLET 3 ILLUSTRATES A HIDDEN CONSEQUENCE. THE TABLE MUST BE SORTED FOR BINARY SEARCHING. WHAT HAPPENS IF THE TABLE HAS BEEN BURNED INTO ROM (READ ONLY MEMORY)? FOR SMALL TABLES, COST OF BURNING NEW ROMS MIGHT BE TOO HIGH, SO THAT A LINEAR SEARCH WITH A SENTINEL MUST BE USED. BUT IF THE TABLE HAS 10,000 ELEMENTS, THEN THE COST OF NOT BURNING NEW ROMS WOULD BE ABSURD.

SELECTING A BETTER ALGORITHM

- BENEFITS OF SELECTING A BETTER ALGORITHM.
  - PROVIDE IMMEDIATE PERFORMANCE IMPROVEMENT.
  - LEAD US TO CONSIDER SPECIAL CASE OF ALGORITHM THAT IMPROVED PERFORMANCE AGAIN.
    - KNUTH'S VERSION IS 4.6 TIMES FASTER THAN THE PREVIOUS BINARY SEARCH.
- BEFORE REPLACING ONE ALGORITHM WITH ANOTHER CONSIDER "HIDDEN" CONSEQUENCES OF REPLACEMENT.
  - DOES THE NEW ALGORITHM MAKE ASSUMPTIONS THAT DO NOT HOLD?
  - CAN THESE ASSUMPTIONS BE MADE TO HOLD.
    - IF SO, IS COST ACCEPTABLE?
    - IS IT MORE COSTLY NOT TO MAKE THE ASSUMPTIONS HOLD?.
- BINARY SEARCH REQUIRES SORTED TABLES.
  - TABLE CAN BE SORTED BEFORE FIRST SEARCH.
  - IS THE COST TOO HIGH?

INSTRUCTOR NOTES

- BOTH SORT TIMES ARE ACCEPTABLE WHEN COMPARED WITH THE LINEAR SEARCHES.

- SUPPOSE WE IMPLEMENT THE INSERTION SORT AND NEED TO SORT A 10,000 ELEMENT TABLE. IF WE USE KNUTH'S BINARY SEARCH, THE SORTING TAKES MORE THAN 3 TIMES AS LONG. RATHER THAN 1.1 MINUTES, THE EFFECTIVE TIME TO DO THE SEARCHING IS 4.5 MINUTES. WE CAN IMPROVE UPON THE ALGORITHM BY USING THE QUICKSORT. ITS EFFECT IS NEGLIGIBLE.

SORTING THE TABLE FOR BINARY SEARCHING

- MANY CHOICES OF SORTING ALGORITHMS. TWO COMMON ONES ARE
  - INSERTION SORT : ORDER (n^2)
  - QUICKSORT : ORDER ($n \log_2 n$)
- BENTLEY DESCRIBES AN INSERTION SORT.
  - SORT TIME : 2.02 n^2 MICROSECONDS.
  - 2 SECONDS FOR 1,000 ELEMENT TABLE.
  - 3.4 MINUTES FOR 10,000 ELEMENT TABLE.
- SEDGEWICK DESCRIBES A QUICKSORT.
  - SORT TIME : 8 $n \log_2 n$.
  - .08 SECONDS FOR 1,000 ELEMENT TABLE.
  - 1.1 SECONDS FOR 10,000 ELEMENT TABLE.
- IF USING KNUTH'S BINARY SEARCH FOR 10,000 TABLE.
  - 1.1 MINUTES FOR 1,000,000 SEARCHES.
  - INSERTION SORT : 3.4 MINUTES.
  - QUICKSORT : 1.1 SECONDS.

INSTRUCTOR NOTES

- THIS EXAMPLE SHOWS THAT USING FIXED POINT TYPES RATHER THAN FLOATING POINT TYPES CAN DECREASE EXECUTION TIME.

- THE NEXT FEW SLIDES DISCUSS THE ADVANTAGES AND DISADVANTAGES OF FIXED POINT/FLOATING POINT TYPES.

```
type Fixed_Type is delta .0001 range 0.0 .. 10.0;

procedure Binary_Search (Item  : in Fixed_Type;
                         Table : in Table_Type;
                         Index : out Natural) is
   Mid_Point : Natural;
   Lower     : Natural := Table'First;
   Upper     : Natural := Table'Last;
begin
   if Lower > Upper then
      Index := 0;
   else
      Mid_Point := (Lower + Upper) / 2;
      if Item = Table (Mid_Point) then
         Index := Mid_Point;
      elsif Item < Table (Mid_Point) then
         Binary_Search (Item, Table (Lower .. Mid_Point - 1), Index));
      else
         Binary_Search (Item, Table (Mid_Point + 1 .. Upper), Index));
      end if;
   end if;
end Binary_Search;
```

- PERFORMANCE

	(FLOAT)	(FIXED)	
WORST CASE = AVERAGE CASE =	23 $\log_2 n$	19 $\log_2 n$	MICROSECONDS/SEARCH

INSTRUCTOR NOTES

- THE FRACTIONAL PART IS ALSO CALLED THE MANTISSA

- EMPHASIZE THAT FLOATING POINT TYPES ARE NEEDED FOR A LARGER RANGE, ------ BUT THAT YOU PAY FOR THIS BY USING MORE COSTLY INSTRUCTIONS
  - IF A PROCESSOR DOES NOT SUPPORT FLOATING POINT, THEN FLOATING POINT ARITHMETIC MUST BE IMPLEMENTED IN SOFTWARE, WHICH IS EVEN MORE COSTLY
  - FLOATING POINT IS STILL EASIER TO USE THAN FIXED POINT WHEN PERFORMING ERROR ANALYSIS ON OPERATIONS MENTIONED, AND IS PROBABLY STILL BEST TO USE AS A FIRST CHOICE

- WE DO NOT DISCUSS MODEL NUMBERS FOR FLOATING POINT, BUT WE WILL FOR FIXED POINT

FLOATING POINT ARITHMETIC

- FLOATING POINT OPERATIONS
  - EVEN WITH HARDWARE SUPPORT, FLOATING POINT OPERATIONS ARE SLOW
  - TYPICAL FORM FOR A FLOATING POINT NUMBER IS

SIGN	EXPONENT	FRACTIONAL PART

    FIRST DIGIT OF FRACTIONAL PART IS NON-ZERO (NORMALIZATION)
  - TYPICAL SEQUENCE OF STEPS IN MULTIPLYING TWO FLOATING POINT NUMBERS
    - ADD EXPONENTS
    - MULTIPLY FRACTION PARTS
    - NORMALIZE PRODUCT

- FLOATING POINT
  - NEEDED FOR LARGE RANGES OF VALUES
  - BETTER THAN FIXED POINT WHEN USING
    - MULTIPLICATION
    - DIVISION
    - MATHEMATICAL FUNCTION : SIN, COS, ETC

    IF PERFORMING ERROR ANALYSIS ON THE RESULTS OF THESE OPERATIONS

INSTRUCTOR NOTES

- THE REASONS FOR THE RESTRICTIONS WILL BE DISCUSSED IN THE NEXT SLIDE, AS WILL THE IMPLEMENTATION OF FIXED POINT TYPES

FIXED POINT ARITHMETIC

- FIXED POINT OPERATIONS
  - TYPICAL FORM FOR A FIXED POINT NUMBER IS

SIGN	INTEGER PART

    WITH THE COMPILER KEEPING TRACK OF THE DECIMAL POINT
  - FIXED POINT OPERATIONS ARE "INTEGER OPERATIONS"
  - RESTRICTIONS ON SOME OPERATIONS

        EXPONENTIATION NOT ALLOWED

        MULTIPLICATION : Left * Right

        - EITHER Left AND Right ARE BOTH FIXED POINT OR ONE IS FIXED POINT AND ONE IS INTEGER
        - IF BOTH ARE FIXED POINT THEN THE PRODUCT MUST APPEAR INSIDE A TYPE CONVERSION

        DIVISION : SIMILAR TO MULTIPLICATION

- FIXED POINT
  - BEST USED FOR SENSORS/COUNTERS
  - BETTER THAN FLOATING POINT WHEN USING

        ADDITION

        SUBTRACTION

    IF PERFORMING ERROR ANALYSIS ON THE RESULTS OF THESE OPERATIONS

INSTRUCTOR NOTES

- DO NOT SPEND TOO MUCH TIME ON THIS SLIDE. THE MAIN POINT IS THAT A MODEL NUMBER HAS THE FORM

  n*T'Small

  THIS WILL BE USED TO EXPLAIN HOW FIXED POINT ARITHMETIC CAN BE IMPLEMENTED AS INTEGER ARITHMETIC

- A FIXED POINT TYPE

  type T is delta D range Low .. High;

  DEFINES A SET OF MODEL NUMBERS

  -(m-1)*T'Small, -(m-2)*T'Small, ..., 0, ... (m-2)*T'Small, (m-1)*T'Small

  WHERE T'Small IS THE LARGEST POWER OF 2 LESS THAN OR EQUAL TO D AND m IS THE SMALLEST POWER OF 2 SUCH THAT

  -m*T'Small $\leq$ Low AND High $\leq$ m*T'Small

- EXAMPLE

  type T is delta .001 range 0.0 .. 2.5

  $$T'Small = \frac{1}{1024} = 2.0**(-10)$$

  M = 2560

  WHICH GIVES MODEL NUMBERS

  -2559/1024, -2558/1024, ..., -1/1024, 0, 1/1024, ..., 2558/1024, 2559/1024

- ANY VALUE IN T IS BETWEEN TWO MODEL NUMBERS

INSTRUCTOR NOTES

- SECOND BULLET

  - THIS SHOULD HELP THE CLASS UNDERSTAND WHY FIXED POINT ARITHMETIC CAN BE IMPLEMENTED AS INTEGER ARITHMETIC

  - IT MIGHT HELP TO GO OVER ONE OR TWO MORE CASES, BUT DO NOT COVER EACH CASE

  - NOTE THAT n AND m ARE SIGNED INTEGERS AS IS I

  - IN THE CASE OF (n*T'Small)/I THE RESULT IS APPROXIMATE, AND IN FACT (n/I)*T'Small IS ITS VALUE. BOTH VALUES ARE BETWEEN THE SAME TWO MODEL NUMBERS.

- A COMPILER MAY REPRESENT THE FIXED POINT NUMBER

  n*T'Small

  AS n WITH THE COMPILER KEEPING TRACK OF T'Small

  - MULTIPLYING BY T'Small FOR OUTPUT OR FOR CONVERSION TO ANOTHER FIXED POINT TYPE
  - DIVIDING BY T'Small FOR INPUT OR FOR CONVERSION FROM ANOTHER NUMERIC TYPE

- FOR A FIXED POINT TYPE T, WE HAVE

  n*T'Small + m*T'Small = (n + m)*T'Small

  n*T'Small - m*T'Small = (n - m)*T'Small

  -(n*T'Small) = (-n)*T'Small

  I*(n*T'Small) = (I*n)*T'Small

  (n*T'Small)*I = (n*I)*T'Small

  (n*T'Small)/I ≈ (n/I)*T'Small

  SO A COMPILER CAN IMPLEMENT

  N*T'Small + m*T'Small

  FOR EXAMPLE, AS

  n+m

INSTRUCTOR NOTES

FIXED POINT ARITHMETIC - CONCLUDED

- FOR FIXED POINT TYPES T1 AND T2

  (n*T1'Small) * (m*T2'Small) = (n+m) * (T1'Small*T2'Small)

  (n*T1'Small) / (m*T2'Small) ≈ (n/m) + (T1'Small/T2'Small)

  THE RESULTS ARE NOT MODEL NUMBERS IN T1 OR T2

  - THE RESULTS ARE REQUIRED TO BE IN A TYPE CONVERSION SO THAT THE DESIRED TARGET TYPE CAN BE DETERMINED

  - THIS IS REQUIRED BY Ada'S STRONG TYPING

INSTRUCTOR NOTES

- THE Inline PRAGMA WAS PRESENTED IN L305, SO BULLET 4 IS JUST A REVIEW. FOR THE INSTRUCTOR, THE PRAGMA CAN APPEAR AT THE PLACE OF A DECLARATIVE ITEM IN A DECLARATIVE PART OR PACKAGE SPECIFICATION, OR AFTER A LIBRARY UNIT IN A COMPILATION.

  - IF IT APPEARS AT A DECLARATIVE ITEM, IT MUST NAME A (GENERIC) SUBPROGRAM DECLARED EARLIER IN THAT SAME DECLARATIVE PART OR PACKAGE SPECIFICATION. IF THE NAME IS OVERLOADED, THE PRAGMA APPLIES TO ALL OF THE SUBPROGRAMS.

  - IF THE PRAGMA APPEARS *AFTER A COMPILATION UNIT*, THE NAME MUST BE THE NAME OF THE LIBRARY UNIT.

- BULLET 5

  IF THE CALLING UNIT IS COMPILED FIRST, THE SUBPROGRAM CANNOT POSSIBLY BE GENERATED INLINE.

- NOTE THAT AN IMPLEMENTATION WILL PROBABLY IGNORE THE INLINE PRAGMA FOR A RECURSIVE PROCEDURE. SLIDE 25-15 DISCUSSES OTHER WAYS FOR DEALING WITH RECURSIVE CALLS.

Inline PRAGMA

- WHEN A SUBPROGRAM CALL APPEARS IN A CRITICAL PATH, PERFORMANCE MAY SOMETIMES BE IMPROVED BY REPLACING THE SUBPROGRAM CALL WITH THE SUBPROGRAM BODY.
- CODING THE SUBPROGRAM BODY INLINE DECREASES PROGRAM READABILITY.
- Ada PROVIDES THE Inline PRAGMA
  - PROGRAMMER WRITES A SUBPROGRAM.
  - COMPILER GENERATES BODY INLINE.
  - PROGRAM READABILITY IS MAINTAINED.
- THE PRAGMA HAS THE FORM:

  pragma Inline (name {, name});

  where *name* is the *name of a subprogram* or generic subprogram.
- COMPILATION DEPENDENCIES MAY INCREASE.
  - IF THE PRAGMA IS APPLIED TO A SUBPROGRAM IN A PACKAGE SPECIFICATION.
    - INLINE EXPANSION IN A UNIT USING THE PACKAGE OCCURS ONLY IF THE PACKAGE BODY IS COMPILED BEFORE THE UNIT.
      - CREATES COMPILATION DEPENDENCY
      - CALLING UNIT MUST BE RECOMPILED WHENEVER SUBPROGRAM IS.
    - IF CALLING UNIT IS COMPILED BEFORE PACKAGE BODY, THEN NO INLINE EXPANSION OCCURS.
  - THE PRAGMA SHOULD ONLY BE USED FOR TUNING.
    - SINCE IT INCREASES RECOMPILATION DEPENDENCIES, IT SHOULD NOT BE USED BEFORE CODE HAS BECOME STABLE.

INSTRUCTOR NOTES

- THIS SLIDE SHOWS A TYPICAL USE OF THE PRAGMA AND A POSSIBLE EXPANSION.
- WE WILL SEE ON THE NEXT SLIDE THE EFFECT OF THIS PRAGMA IN A CRITICAL CODE SECTION.

A SWAP PROCEDURE

```
procedure Swap (Value_1, Value_2 : in out Float) is

   Temp : Float;

begin

   Temp := Value_1;
   Value_1 := Value_2;
   Value_2 := Temp;

end Swap;
pragma Inline (Swap);
```

A CALL SUCH AS

```
   Swap (A, B)
```

IS COMPILED AS IF THE FOLLOWING HAD BEEN CODED AT THE PLACE OF THE CALL.

```
   declare
      Temp : Float;
   begin
      Temp := A;
      A := B;
      B := Temp;
   end; -- block
```

INSTRUCTOR NOTES

- STRESS THAT THIS PRAGMA SHOULD BE USED ONLY WHEN IN A CRITICAL PATH.

- INSERTION SORTS WERE PRESENTED IN L305. THE SORT WORKS BY SUCCESSIVELY INSERTING THE Ith ELEMENT IN ITS PROPER PLACE AMONG THE FIRST I-1 ELEMENTS. DO NOT GO INTO MUCH DETAIL ABOUT THIS SORT. JUST EXPLAIN THAT IT USES Swap TO EXCHANGE ELEMENTS.

USING THE SWAP PROCEDURE

```
procedure Insertion_Sort (Table : Table_Type) is
   J : Natural;
begin
   for I in 2 .. Table'Last loop
       J := I;
       while J > 1 and then Table (J) < Table (J - 1) loop
          Swap (Table (J), Table (J - 1));
          J := J - 1;
       end loop;
   end loop;
end T;
```

- EXECUTION TIME

  - 6.00 N^2 MICROSECONDS WITHOUT INLINE PRAGMA.
  - 3.90 N^2 MICROSECONDS WITH INLINE PRAGMA.
  - 35% SPEEDUP.

- IF SORTING IS ONLY PERFORMED ONCE ON 100 ELEMENTS.

  - SAVING ONLY .06 SECONDS - .039 SECONDS = .021 SECONDS.
  - INLINE EXPANSION IS OF NO VALUE.

- IF SORTING 10,000 ELEMENT

  - SAVING 600 SECONDS - 390 SECOND = 210 SECONDS = 3.5 MINUTES.
  - INLINE EXPANSION IS OF VALUE.

INSTRUCTOR NOTES

- A SMART COMPILER WOULD HAVE MOVED (P1/2)**2 OUTSIDE OF THE LOOP. WE WILL LOOK AT SUCH THINGS IN THE NEXT SECTION.

- IN GENERAL, A COMPILER CANNOT MOVE THE FUNCTION CALL OUT BECAUSE OF POSSIBLE SIDE EFFECTS - MODIFYING A GLOBAL VARIABLE.

- IN GENERAL, IF A SUBPROGRAM, APPEARING IN A CRITICAL LOOP, ALWAYS RETURNS THE ANSWER FOR THE SAME INPUT, THEN A CALL TO SUCH A SUBPROGRAM CAN BE REMOVED FROM A LOOP IF IT IS ALWAYS CALLED WITH THE SAME INPUTS.

REMOVING SUBPROGRAM CALLS FROM LOOPS

- PERFORMANCE IMPROVEMENTS CAN BE GAINED BY MOVING SUBPROGRAM CALLS FROM LOOPS.

```
for I in Table'Range loop
   Table (I) := Table (I) * Exp ((Pi/2)**2);
end loop;
```

EXECUTION TIME: 138N MICROSECONDS (WHERE N = Table'Length)

```
Multiplier := Exp ((Pi/2)**2);
for I in Table'range loop
   Table (I) := Table (I) * Multiplier;
end loop;
```

EXECUTION TIME: 7.9 N MICROSECONDS (WHERE N=Table'Length)

INSTRUCTOR NOTES

- RECURSIVE SUBPROGRAMS ARE GENERALLY EASIER TO UNDERSTAND UNLESS THEY APPEAR IN CRITICAL CODE PIECES.

- WHEN PERFORMANCE IS AN ISSUE, A SIMPLE TYPE OF RECURSION, TAIL RECURSION, CAN BE EASILY REMOVED.

- TAIL RECURSION OCCURS WHEN THE LAST ACTION A SUBPROGRAM PERFORMS IS TO CALL ITSELF RECURSIVELY. IN THIS CASE THESE CALLS CAN BE REPLACED BY A LOOP THAT REPEATS THE BEGINNING OF THE SUBPROGRAM. LOCAL VARIABLES KEEP TRACK OF THE PARAMETER VALUES.

- THE EXAMPLE IS THE Binary_Search PRESENTED EARLIER.

REMOVING RECURSIVE CALLS

```
procedure Binary_Search (Item  : in Float;
                         Table : in Table_Type;
                         Index : out Natural) is

   Mid_Point : Natural;
   Lower : Natural := Table'First;
   Upper : Natural := Table'Last;
begin
   if Lower > Upper : then
       Index := 0;
   else
       Mid_Point := (Lower + Upper) / 2;
       if Item = Table (Mid_Point) then
          Index := Mid_Point;
       elsif Item < Table (Mid_Point) then
          Binary_Search (Item, Table (Lower .. Mid_Point -1), Index));
       else
          Binary_Search (Item, Table (Mid_Point +1.. Upper), Index));
       end if;
   end if;
end Binary_Search;
```

- EXAMPLE OF TAIL RECURSION.
  - EVERY RECURSIVE CALL OCCURS JUST BEFORE RETURNING.
- RECURSION CAN BE REMOVED.
  - INSTEAD OF SUBPROGRAM CALL, GO TO THE BEGINNING OF THE SUBPROGRAM.
  - USE LOCAL VARIABLES TO HOLD PARAMETER VALUES.

INSTRUCTOR NOTES

- VERTICAL BARS ON LEFT HAND SIDE ARE CHANGE BARS.

- THE GOTO IS USED TO ILLUSTRATE WHAT IS HAPPENING. A VERSION WITH A LOOP IS SHOWN NEXT.

```
procedure Binary_Search (Item  : in Float;
                         Table : in Table_Type;
                         Index : out Natural) is
   Mid-Point : Natural;
   Lower     : Natural := Table'First;
   Upper     : Natural := Table'Last;

begin
   <<Start>>
       if Lower > Upper then
          Index := 0;
          return;
       else
          Mid_Point := (Lower + Upper) /2;
          if Item = Table (Mid_Point) then
              Index := Mid_Point;
              return;
          elsif Item < Table (Mid_Point) then
             Upper := Mid_Point -1;
             goto Start;
          else
             Lower := Mid_Point + 1;
             goto Start;
          end if;
       end if;
end Binary_Search;
```

INSTRUCTOR NOTES

- THIS VERSION IS A LITTLE MORE COMPLEX THAN THE RECURSIVE VERSION. BUT IF IMPROVED PERFORMANCE IS NEEDED, THIS IS APPROPRIATE.

- NOW THAT THE RECURSION HAS BEEN ELIMINATED, THE INLINE PRAGMA COULD BE USED IF THAT WAS APPROPRIATE.

TAIL RECURSION REMOVED

```
procedure Binary_Search (Item  : in Float;
                         Table : in Table_Type;
                         Index : out Natural) is
   Mid_Point : Natural;
   Lower : Natural := Table'First;
   Upper : Natural := Table'Last;
begin
   loop
      if Lower > Upper then
         Index := 0;
         exit;
      else
         Mid_Point := (Lower + Upper)/2;
         if Item    = Table (Mid_Point) then
            Index  := Mid_Point;
            exit;
         elsif Item < Table (Mid_Point) then
            Upper := Mid_Point - 1;
         else
            Lower := Mid_Point +1;
         end if;
      end if;
   end loop;
end Binary_Search;
```

- EXECUTION TIME

  - RECURSIVE VERSION - 23 $Log_2 n$
  - NON-RECURSIVE VERSION - 16 $log_2 n$
  - 30% SPEEDUP

INSTRUCTOR NOTES

- FIBONACCI NUMBERS HAVE BEEN USED FOR SORTING.
- THIS VERSION OF Fib PERFORMS NEEDLESS RECOMPUTATION. EACH TIME Fib (25) IS CALLED, THE FIRST 25 FIBONACCI NUMBERS WILL BE RECOMPUTED.
- THERE ARE TWO STRAIGHTFORWARD WAYS TO CORRECT THIS.

- FIBONACCI NUMBERS:

  1,1,2,3,5,8,13,21,34,55,89, ...
  Fib(0) = Fib (1) = 1
  Fib(n) = Fib (n-1) + Fib (n-2), n 2

```
subtype Fibonacci_Range is Natural range 0 .. Max_Fibonacci;
function Fib (N : Fibonacci_Range) return Positive is

   A : Fibonacci_Range := 1;
   B : Fibonacci_Range := 1;
   C : Fibonacci_Range;
begin
   if N <= 1 then
       return 1;
   else
       for I in 2 .. N loop
          C := A + B;
          A := B;
          B := C;
       end loop;
       return C;
   end if;
end Fib;
```

- GIVEN TWO CONSECUTIVE FIBONACCI NUMBERS, ALL LARGER FIBONACCI NUMBERS CAN BE CALCULATED.

INSTRUCTOR NOTES

- RATHER THAN RECOMPUTING EACH TIME, ALL THE FIBONACCI NUMBERS UP TO SOME MAXIMUM ARE COMPUTED JUST ONCE. THIS CAN TAKE PLACE DURING PACKAGE ELABORATION, OR IF ONLY THE FIRST "FEW" FIBONACCI NUMBERS ARE NEEDED, THEY CAN BE SPECIFIED AT COMPILE TIME.

- ALL THAT Fib (N) DOES NOW IS SELECT THE STORED VALUE OF THE Nth FIBONACCI NUMBER. IF IN ADDITION, THE CALL IS IN A CRITICAL PATH, THE Inline PRAGMA CAN BE USED TO REDUCE THE CALL TO SELECTING AN ARRAY COMPONENT.

STORING PRECOMPUTED RESULTS

```
package Fibonacci_Package is

   Max_Fibonacci : constant := ---;
   subtype Fibonacci_Range is Positive range 1  .. Max_Fibonacci;

   function Fib (N : Fibonacci_Range) return Positive;
end Fibonacci_Package;

package body Fibonacci_Package is

   Numbers : array (Fibonacci_Range) of Positive;

   function Fib (N : Fibonacci_Range) return Positive is
   begin
       return Numbers (N);
   end Fib;

begin -- Fibonacci_Package

   Numbers (1) := 1;
   Numbers (2) := 1;
   for I in 3 .. Max_Fibonacci loop
       Numbers (I) := Numbers (I - 1) + Numbers (I - 2);
   end loop;

end Fibonacci_Package;
```

- IF Max_Fibonacci IS "SMALL," Numbers CAN BE GIVEN AN INITIAL VALUE AT COMPILATION TIME INSTEAD OF WHEN THE PACKAGE IS ELABORATED.

```
Numbers:  constant array (Fibonacci_Range) of Positive :=
          (1,1,2,3,5,8,13,21,34,55,87,142,229,371,600,971);
```

INSTRUCTOR NOTES

- IF A LARGE RANGE OF FIBONACCI NUMBERS MUST BE SUPPORTED, THEN AN IN-BETWEEN APPROACH CAN BE USED. ONLY COMPUTE THE NUMBER WHEN IT IS NEEDED, AND STORE EACH INTERMEDIATE NUMBER CALCULATED. KEEP TRACK OF THE LARGEST NUMBER THAT HAS BEEN COMPUTED (Top).

- WHEN THE NTH FIBONACCI NUMBER IS REQUESTED, CHECK IF N <= Top. IF IT IS, THEN THE NTH FIBONACCI NUMBER HAS ALREADY BEEN CALCULATED ITS VALUE IS Numbers (N). OTHERWISE, THE NTH FIBONACCI NUMBER HAS NOT BEEN CALCULATED. DO SO, BY STARTING WITH Numbers (Top) AND Numbers (Top -1), SAVING THE INTERMEDIATE RESULTS.

- THE PROBLEM IS BEING DYNAMICALLY SIMPLIFIED SINCE TOP CONTINUES TO INCREASE, AND FOR N < Top, Fib (N) IS A SIMPLE TABLE LOOKUP.

- FILL IN TABLE ENTRIES AS THE NEED ARISES:

```
package body Fibonacci_Package is

   Numbers : array (Fibonacci_Range) of Positive;
   Top     : Fibonacci_Range := 2;

   function Fib (N : Fibonacci_Range) return Positive is
   begin
       if N > Top then
          for I in Top + 1 .. N loop
              Numbers (I) := Numbers (I - 1) + Numbers (I - 2);
          end loop;
       end if;
       Top := N;
       return Numbers (N);
   end Fib;

begin -- Fibonacci_Package

   Numbers (1) := 1;
   Numbers (2) := 1;

end Fibonacci_Package;
```

- DYNAMIC EXAMPLE OF BENTLEY'S RULE OF PROBLEM SIMPLIFICATION TO INCREASE THE EFFICIENCY OF A PROGRAM, SIMPLIFY THE PROBLEM IT SOLVES.

INSTRUCTOR NOTES

- A LOOP EXECUTING ITS STATEMENTS 10 TIMES CAN BE UNFOLDED TO REMOVE THE LOOP OVERHEAD.
- THIS APPROACH ONLY WORKS FOR LOOPS OVER FIXED RANGES.
- THE NEXT SLIDE EXTENDS THIS TO VARIABLE RANGES.

COMPLETE LOOP UNFOLDING FOR FIXED RANGES

```
Sum := 0.0;
for I in 1 .. 10 loop
   Sum := Sum + Table (I);
end loop;
```

- EXECUTION TIME: 63.4 MICROSECONDS

```
Sum  := Table (1) + Table (2) + Table (3) + Table (4) + Table (5)
      + Table (6) + Table (7) + Table (8) + Table (9) + Table (10);
```

- EXECUTION TIME: 22.1 MICROSECONDS

  - ELIMINATED ADDING 1 TO I
  - ELIMINATED COMPARING I to 10.
  - ELIMINATED BRANCHING
  - REPLACED RUNTIME CALCULATION OF ARRAY-COMPONENT ADDRESSES WITH COMPILE-TIME CALCULATION (COMPILER OPTIMIZATION)

INSTRUCTOR NOTES

- THE DOUBLE UNFOLDING CUTS THE LOOP OVERHEAD IN HALF.

- WE COULD UNFOLD MORE THAN TWICE, SAY 5 TIMES BY WRITING

```
Sum := Table (I) + Table (I + 1) + Table (I + 2) + Table (I + 3) +
       Table (I+4);
I := I + 5;
```

AND MODIFYING THE PREPROCESSING CODE (THE IF STATEMENT) TO ENSURE THE LOOP ALWAYS HAS A MULTIPLE OF 5 ELEMENTS TO WORK WITH.

DOUBLE UNFOLDING FOR VARIABLE LOOP RANGE

```
function Table_Sum (Table : Table_Type) return Float is
   Sum : Float := 0.0;
begin
   for I in 1 .. Table'Last loop
       Sum := Sum + Table (I);
   end loop;
   return Sum;
end Table_Sum;
```

- DOUBLE UNFOLDING OF LOOP
- ADD TWICE FOR EACH COMPARE.
- MUST HANDLE ODD NUMBER OF ELEMENTS.

```
function Table_Sum (Table : Table_Type) return Float is
   Sum : Float;
   I : Positive;
begin
   if Table'Last mod 2 = 0 then -- even number of elements
       Sum := 0.0;
       I := 1;
   else -- odd number of elements
       Sum := Table (1);
       I := 2;
   end if;

       -- EVEN NUMBER OF ELEMENTS LEFT TO PROCESS.

   while I < Table'Last loop
       Sum := Sum + Table (I) + Table (I + 1);
       I := I + 2;
   end loop;
   return Sum;
end Table_Sum;
```

INSTRUCTOR NOTES

- JUST IN CASE SOMEONE ASKS, IF A SUBPROGRAM IS CALLED MANY TIMES AND THE Inline PRAGMA IS NOT APPROPRIATE, THE Elaboration_Check MIGHT BE SUPPRESSED. THIS SHOULD ONLY BE DONE WHEN IT IS SAFE. MOREOVER, THIS CHECK MIGHT BE PERFORMED BY THE COMPILER PRIOR TO RUNTIME.

RUNTIME CHECKS

- Ada COMPILERS MAY GENERATE CODE TO CHECK FOR CERTAIN CONDITIONS THAT REQUIRE AN EXCEPTION TO BE RAISED. SOME OF THE CHECKS ARE NAMED.
- THE Constraint_Error EXCEPTION HAS THE NAMED CHECKS
  - Access_Check: CHECK THAT WHEN ACCESSING A SELECTED COMPONENT, AN INDEXED COMPONENT, ETC. OF AN OBJECT DESIGNATED BY AN ACCESS VALUE, THAT THE ACCESS VALUE IS NOT NULL.
  - Discriminant_Check: CHECK THAT DISCRIMINANT CONSTRAINTS ARE SATISFIED.
  - Index_Check: INCLUDES WHEN ACCESSING A COMPONENT OF AN ARRAY OBJECT, CHECK FOR EACH DIMENSION THAT THE GIVEN INDEX VALUE BELONGS TO THE RANGE DEFINED BY THE BOUNDS OF THE ARRAY.
  - Length_Check: CHECK THAT THERE IS A MATCHING COMPONENT FOR EACH COMPONENT OF AN ARRAY IN THE CASE OF ARRAY ASSIGNMENTS, TYPE CONVERSIONS, AND LOGICAL OPERATORS FOR ARRAYS OF BOOLEAN COMPONENTS.
  - Range_Check: INCLUDES CHECKING THAT A VALUE SATISFIES A RANGE CONSTRAINT.
- THE Numeric_Error EXCEPTION HAS THE NAMED CHECKS.
  - Division_Check: CHECK THAT SECOND OPERAND IS NOT ZERO FOR THE OPERATIONS /, REM AND MOD.
  - Overflow_Check: CHECK THAT THE RESULT OF A NUMERIC OPERATION DOES NOT OVERFLOW.
- THE Program_Error EXCEPTION HAS THE NAMED CHECK.
  - Elaboration_Check: WHEN EITHER A SUBPROGRAM IS CALLED, A TASK ACTIVATED, GENERIC INSTANTIATION ELABORATED, CHECK THAT BODY OF CORRESPONDING UNIT HAS ALREADY BEEN ELABORATED.
- THE Storage_Error EXCEPTION HAS THE NAMED CHECK.
  - Storage_Check: CHECK THAT ENOUGH SPACE LEFT IN COLLECTION FOR ALLOCATOR EXECUTION; ENOUGH SPACE LEFT FOR SUBPROGRAM OR TASK.

INSTRUCTOR NOTES

THE SUPPRESS PRAGMA SAVES TIME BUT DEPENDS ON WHAT YOU SUPPRESS AND WHAT PERCENT OF THE CODE USES THE CHECK YOU'RE SUPPRESSING.

Suppress PRAGMA

- THE CHECKS ON THE PREVIOUS SLIDE CAN BE SUPPRESSED.

  pragma Suppress ([name of check] [, [ON =>] [name]]);

- [name] DEPENDS ON THE EXCEPTION THAT THE CHECK BELONGS TO

  - Constraint_Error : OBJECT OR TYPE NAME
  - Numeric_Error : NUMERIC TYPE NAMES
  - Program_Error : NAME OF TASK UNIT, GENERIC UNIT OR SUBPROGRAM
  - Storage_Error : NAME OF ACCESS TYPE, TASK UNIT OR SUBPROGRAM.

- THE PRAGMA MAY APPEAR

  - WITHIN A DECLARATIVE PART

    - PERMISSION TO OMIT CHECK EXTENDS FROM THE PLACE OF THE PRAGMA TO THE END OF THE DECLARATIVE REGION CONTAINING THE DECLARATIVE PART.

  - WITHIN A PACKAGE SPECIFICATION.

    - ONLY APPLIES TO ENTITY DECLARED WITHIN PACKAGE SPECIFICATION.

    - PERMISSION TO OMIT CHECK EXTENDS FROM THE PLACE OF THE PRAGMA TO THE END SCOPE OF THE ENTITY.

- IF [name] IS SPECIFIED:

  - CHECK IS OMITTED ONLY FOR ENTITY NAMED.
  - [name] CAN BE SIMPLE OR EXPANDED NAME.

INSTRUCTOR NOTES

- EMPHASIZE THAT THE Suppress PRAGMA SHOULD BE VIEWED AS A CONTRACT BETWEEN THE PROGRAMMER AND THE COMPILER.

- THE Suppress PRAGMA SHOULD ONLY BE USED WHEN IT IS SAFE TO DO SO.

- THE PROGRAMMER IS TELLING THE COMPILER THAT A SPECIFIC CONDITION CANNOT OCCUR, THEREFORE THE CHECK IS NOT NEEDED AND SHOULD NOT BE GENERATED.

- PROGRAMS THAT CONTAIN A SUPPRESS PRAGMA FOR A CONDITION THAT DOES OCCUR, ARE ERRONEOUS.

INSTRUCTOR NOTES

- THE ASSUMPTION IS THAT THE COMPUTATION OCCURS IN A CRITICAL PATH.

- GENERALLY, CHECKS SHOULD NOT BE SUPPRESSED UNLESS THEY OCCUR IN CRITICAL PATHS, AND THEN ONLY IF IT CAN BE ESTABLISHED THAT THE CHECK IS NOT NEEDED.

- IT IS A WELL KNOWN MATHEMATICAL PROPERTY THAT

  $\sin^2 X + \cos^2 X = 1$

- SUPPOSE WE HAVE A COSINE FUNCTION

```
function Cos (X : Angle_Type) return Trig_Type;
```

WHERE WE HAVE THE DEFINITION

```
subtype Trig_Type is Float digits 6 range -1.0 .. 1.0;
```

- TO COMPUTE SINE WE CAN WRITE

```
Sin : Trig_Type;
pragma Suppress (Range_Check, On => Sin);
   ...
Sin := (1.0 - Cos(X)**2)**(-2);
```

SINCE WE KNOW THE RESULT WILL BE WITHIN RANGE.

INSTRUCTOR NOTES

- FORMAL VERIFICATION CAN BE USED TO ESTABLISH THAT A CHECK IS NOT NEEDED.

- EVEN IF FORMALLY VERIFIED, THERE IS NO NEED TO SUPPRESS Index_Check ON Table UNLESS THE PERFORMANCE IMPROVEMENT IS NEEDED.

Index_Check

```
procedure Binary_Search (Item  : in Float;
                         Table : in Table_Type;
                         Index : out Natural) is
   pragma Suppress (Index_Check, On => Table);
   Lower : Natural := 1;
begin
   if Table (512) < Item then
      Lower := 1000 + 1 - 512;
   end if;
   if Table (Lower + 256) < Item then
      Lower := Lower + 256;
   end if;
   if Table (Lower + 128) < Item then
      Lower := Lower + 128;
   end if;
   if Table (Lower + 64) < Item then
      Lower := Lower + 64;
   end if;
   if Table (Lower + 32) < Item then
      Lower := Lower + 32;
   end if;
   if Table (Lower + 16) < Item then
      Lower := Lower + 16;
   end if;
   if Table (Lower + 8) < Item then
      Lower := Lower + 8;
   end if;
   if Table (Lower + 4) < Item then
      Lower := Lower + 4;
   end if;
   if Table (Lower + 2) < Item then
      Lower := Lower + 2;
   end if;
   if Table (Lower + 1) < Item then
      Lower := Lower + 1;
   end if;
   if Lower <= 1000 and then Table (Lower) = Item then
      Index := Lower;
   else
      Index := 0;
   end if;
end Binary_Search;
```

- IT CAN BE VERIFIED THAT EVERY COMPONENT ACCESS TO Table HAS AN INDEX EXPRESSION THAT IS IN BOUNDS.

INSTRUCTOR NOTES

- ASSEMBLY LANGUAGE CODING OF SELECTIVE PROGRAM PIECES SHOULD NOT BE IGNORED.
- THE Interface_Pragma WAS DISCUSSED IN L305.

RECODING IN ASSEMBLY LANGUAGE

- IF ALL ELSE FAILS, ASSEMBLY LANGUAGE CODING OF SELECTIVE PROGRAM PIECES CAN BE CONSIDERED.
  - CAN EXPLOIT MACHINE INSTRUCTIONS
  - SACRIFICES CLARITY
  - REQUIRES KNOWLEDGE OF COMPILER DATA MAPPING
  - INCREASES COMPILATION DEPENDENCIES
  - SHOULD ONLY BE USED WHEN IMPROVEMENT CANNOT BE ACHIEVED WITHOUT IT

- KNUTH'S BINARY SEARCH HAS EXECUTION TIME 5 $\log_2 n$.
  - 1.1 MINUTES FOR 10,000 ELEMENT TABLE

- RECODING IN ASSEMBLY LANGUAGE RESULTED IN EXECUTION TIME 0.9 $\log_2 n$
  - 12 SECONDS FOR 10,000 ELEMENT TABLE

- SUBPROGRAMS WRITTEN IN OTHER LANGUAGES CAN BE MADE AVAILABLE TO AN Ada PROGRAM BY THE Interface PRAGMA

  pragma Interface (language name, subprogram name);

- PROGRAMING IN ASSEMBLY LANGUAGE
  - HINDERS PORTABILITY
  - SHOULD ONLY BE CONSIDERED FOR THE 3% OF THE PROGRAM THAT IS EXECUTED MOST OFTEN.

INSTRUCTOR NOTES

COMPARING THE EXTREMES

- TO PERFORM ONE MILLION SEARCHES ON A 10,000 ELEMENT TABLE
  - STARTED WITH LINEAR SEARCH; 10.1 HOURS
  - ENDED UP WITH ASSEMBLER VERSION OF BINARY SEARCH: 13.1 SECONDS
    - 12 SECONDS SEARCHING
    - 1.1 SECONDS SORTING
- TUNING RESULTED IN A FACTOR 2,800 SPEED UP.

INSTRUCTOR NOTES

A PROPOSED RUSSIAN EXTENSION

pragma Suppress (Czechs)

INSTRUCTOR NOTES

- ALLOW 45 MINUTES FOR THIS SECTION.

- THE POINT OF THIS SECTION IS THAT A SMART OPTIMIZING COMPILER CAN PRODUCE EFFICIENT PROGRAMS, THEREFORE PROGRAMS SHOULD BE WRITTEN WITH CLARITY IN MIND.

Section 26

WHAT'S BEST LEFT TO THE COMPILER

INSTRUCTOR NOTES

- THESE TARGET-MACHINE INDEPENDENT OPTIMIZATIONS ARE EXPLAINED IN THIS SECTION. WE WILL NOT CONCENTRATE ON MACHINE DEPENDENT OPTIMIZATIONS.

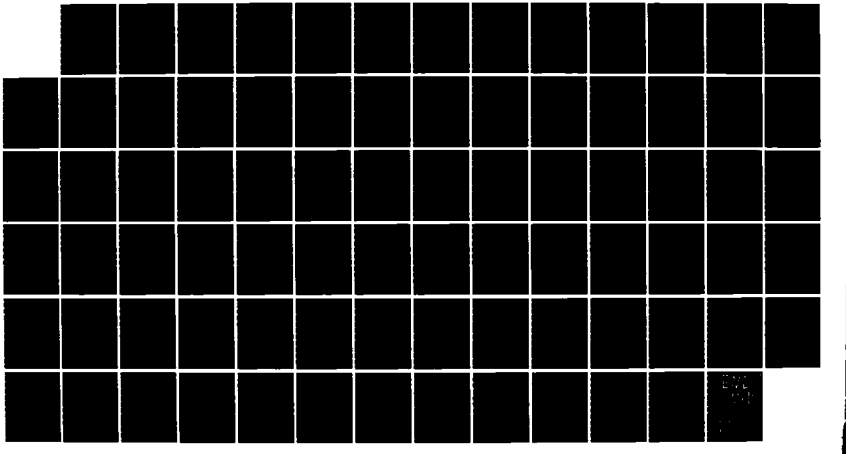
- BULLET 2 - THE COMPILER "ADDS" THE RUNTIME CHECKS TO THE PROGRAM BEING COMPILED. IT THEN USES OPTIMIZATIONS SUCH AS THE ONES DESCRIBED IN THIS SECTION TO PERFORM THE RUNTIME CHECK AT COMPILE TIME.

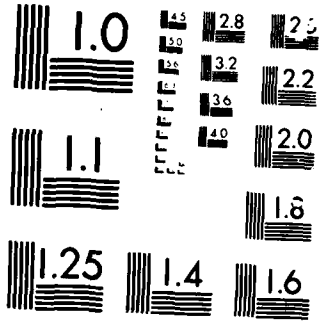
LETTING THE COMPILER HELP

- OPTIMIZING COMPILERS PERFORM TWO TYPES OF OPTIMIZATIONS
  - TARGET MACHINE INDEPENDENT
    - TASKING IDIOMS (STYLIZED USES OF Ada TASKS TO BUILD PROTOCOLS SUCH AS MONITORS)
    - COMMON SUBEXPRESSIONS
    - CODE MOTION
    - STRENGTH REDUCTION
    - LOOP JAMMING
    - DEAD CODE ELIMINATION
    - ETC.
  - TARGET MACHINE DEPENDENT
    - EXPLOIT SPECIAL INSTRUCTIONS (E.G. USE SHIFT INSTRUCTIONS FOR MULTIPLYING A Natural VALUE BY A POWER OF 2)
    - REGISTER USAGE
    - ETC.
- MANY OF THESE OPTIMIZATIONS ALLOW RUNTIME CHECKS TO BE PERFORMED AT COMPILE-TIME.
- A "SMART" OPTIMIZING COMPILER LETS THE PROGRAMMER WRITE CLEAR, EASILY MAINTAINABLE PROGRAMS THAT ARE ALSO EFFICIENT.

AD-A166 352 ADA (TRADE NAME) TRAINING CURRICULUM REAL-TIME SYSTEMS IN ADA L401 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 6/6

UNCLASSIFIED F/G 5/9 NL





MICROCOPY RESOLUTION TEST CHART

INSTRUCTOR NOTES

- WE WILL DISCUSS THREE OF THESE PATTERNS.

THE ROLE OF IMPLEMENTATION DEFINED PRAGMAS IN OPTIMIZATION

- PRAGMAS GIVE "ADVICE" TO THE COMPILER.
  - Inline
  - Pack
- AN IMPLEMENTATION CAN DEFINE ITS OWN PRAGMAS.
- IMPLEMENTATION DEFINED PRAGMAS MUST NOT CHANGE THE LEGALITY OF A PROGRAM.
- PRAGMAS CAN BE USED TO ALERT THE COMPILER TO SPECIAL PATTERNS IN THE PROGRAM FOR WHICH IT CAN GENERATE MORE EFFICIENT CODE.
- IT MAY BE INEFFICIENT FOR THE COMPILER TO SEARCH FOR CERTAIN PATTERNS.

INSTRUCTOR NOTES

- THE TASK OPTIMIZATIONS GENERALLY NEED PRAGMAS.
- BULLET 5 - THE COMPILER MAY GENERATE AN ERROR MESSAGE IF THE ADVICE GIVEN BY THE PRAGMA IS FALSE.

TASKING OPTIMIZATIONS

- COMMONLY OCCURRING SPECIAL PATTERNS OF TASK INTERACTIONS FOR WHICH BETTER IMPLEMENTATIONS ARE AVAILABLE THAN THE IMPLEMENTATION OF THE GENERAL CASE.

- PRAGMAS PROBABLY NEEDED.

- SEE PAPERS BY
  - P. HILFINGER
  - A. N. HABERMANN AND I. R. NASSI

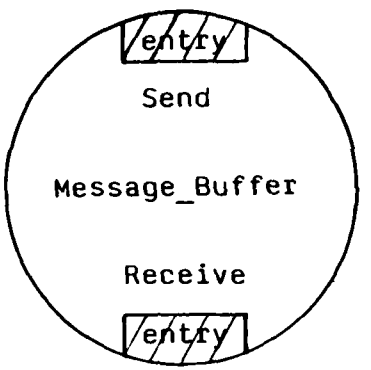
INSTRUCTOR NOTES

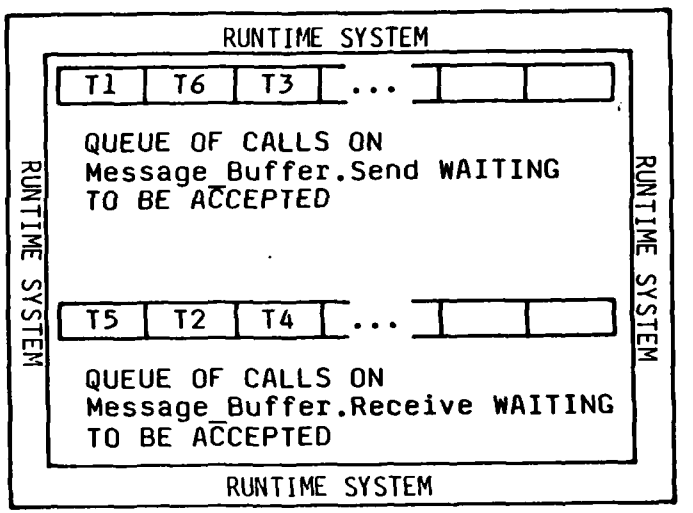
- REMIND THE CLASS THAT EACH ENTRY HAS A QUEUE ASSOCIATED WITH IT.
- THE QUEUE IS NEEDED BECAUSE ONE OF SEVERAL TASKS COULD CALL THE ENTRIES OF THE MESSAGE BUFFER.

TASK OPTIMIZATION 1: QUEUES

- CONSIDER A MESSAGE BUFFER

```
task type Message_Buffer_Type is
   entry Send (Message : in Message_Buffer_Type);
   entry Receive (Message : out Message_Buffer_Type);
end Message_Buffer_Type;
Message_Buffer : Message_Buffer_Type;
```





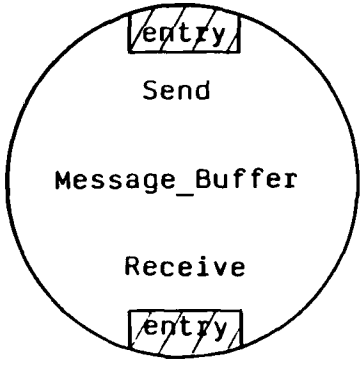
- AN ENTRY QUEUE IS NEEDED TO KEEP TRACK OF TASKS WAITING TO RENDEZVOUS AT THAT ENTRY. (THESE QUEUES ARE MAINTAINED BY THE RUNTIME SYSTEM AND HIDDEN FROM THE PROGRAMMER.)

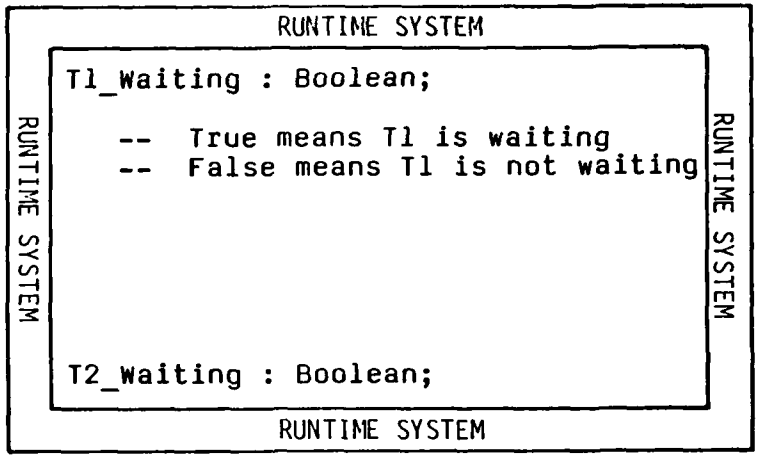
INSTRUCTOR NOTES

- MANY TIMES A MESSAGE BUFFER IS USED ONLY TO PREVENT A PRODUCER TASK FROM BEING SLOWED DOWN BY A CONSUMER TASK. IN SUCH A CASE, THERE IS A SINGLE TASK - A PRODUCER - CALLING THE Send ENTRY AND A SINGLE TASK - A CONSUMER - CALLING THE Receive ENTRY. THEREFORE A QUEUE IS NOT NEEDED TO KEEP TRACK OF THE TASKS CALLING THE MESSAGE BUFFER.

TASK OPTIMIZATION 1: QUEUES (CONTINUED)

- SUPPOSE ONLY ONE TASK (T1) CALLS Send AND ONLY ONE TASK (T2) CALLS Receive.





- SINCE ONLY ONE TASK, T1, CAN CALL Send, A QUEUE IS NOT NEEDED. ALL THAT IS NEEDED IS AN INDICATION OF WHETHER OR NOT A TASK IS WAITING FOR A RENDEZVOUS WITH SEND. IF THERE IS ONE, IT MUST BE T1.

- THE Message_Buffer TASK NEEDS NO QUEUES FOR ITS ENTRIES.

- A COMPILER CANNOT ALWAYS DETECT THIS.
  - IT MIGHT REQUIRE HELP THROUGH A PRAGMA.

INSTRUCTOR NOTES

- ALTHOUGH WE HAVE NOT SHOWN IT, THE User AND Server TASKS CAN ENGAGE IN OTHER RENDEZVOUS.

- IF THE User TASK IS ACCEPTING TIMED ENTRY CALLS, IT WILL WANT TO MINIMIZE THE TIME IT SPENDS BETWEEN ACCEPT STATEMENTS. IT CAN USE AGENT TASKS TO ACCOMPLISH THIS.

- THE CONCURRENCY BETWEEN Agent AND Server IS NOT REQUIRED. IN FACT, IF Server WAS FAST ENOUGH, Agent WOULD NOT BE NEEDED. *THEREFORE, IT IS SUFFICIENT FOR* Agent AND Server TO COMMUNICATE THROUGH SIMPLE COROUTINE CALLS.

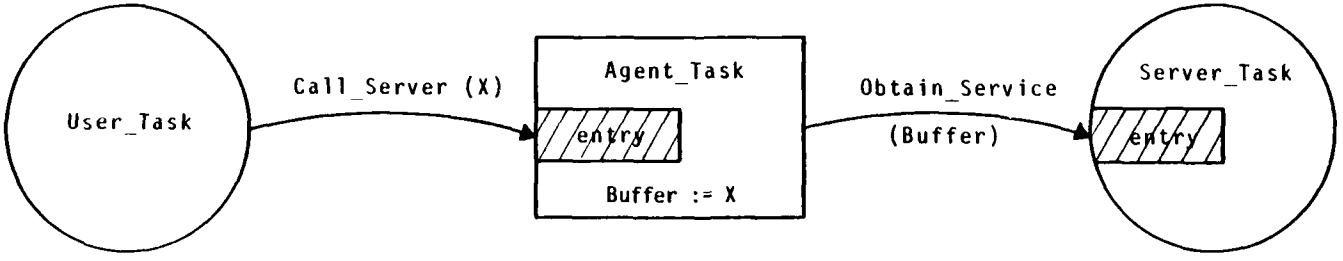
TASK OPTIMIZATIONS 2: AGENT TASKS - BACKGROUND

- A PROGRAM MIGHT CONTAIN AN AGENT TASK OF THE FORM

```
task Agent_Task is
   entry Call_Server (X : in Data_Type);
end Agent_Task;
```

```
task body Agent_Task is
   Buffer : Data_Type;
begin
   loop
      accept Call_Server (X : in Data_Type) do
         Buffer := X;
      end Call_Server;
      Server_Task.Obtain_Service (Buffer);
   end loop;
end Agent_Task;
```

- A TASK USING THE SERVICES OF Server_Task CAN CALL Agent_Task.Call_Server AND PROCEEDS WITHOUT WAITING FOR Server_Task TO REACH AN ACCEPT STATEMENT.



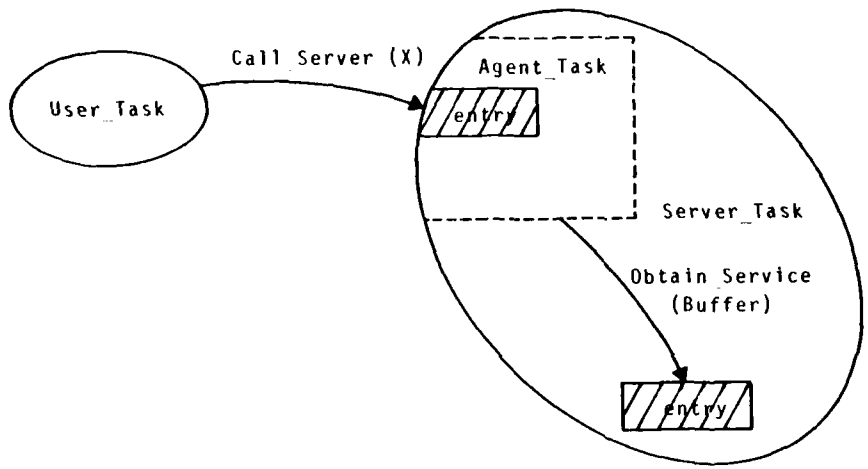
- SAME PRINCIPLE AS A MESSAGE BUFFER.

INSTRUCTOR NOTES

- Agent AND Server ARE NOW EXECUTED BY THE SAME VIRTUAL PROCESSOR.

TASK OPTIMIZATIONS 2: AGENT TASKS - CONCLUDED

- IMPLEMENTATION OF TASK INTERACTION CAN BE GREATLY SIMPLIFIED.



- A CALL ON Agent_Task.Call_Server IS COMPILED TO DIRECTLY QUEUE A CALL ON Server_Task.Obtain_Service, AND THEN FINISH.
- Agent_Task DOES NOT REALLY EXIST IN OBJECT CODE.
- NO NEED FOR CONTEXT SWITCHES INVOLVING Agent_Task
  - SAVING STATE OF TASK BEING SUSPENDED
  - RESTORING STATE OF TASK BEING AWAKENED

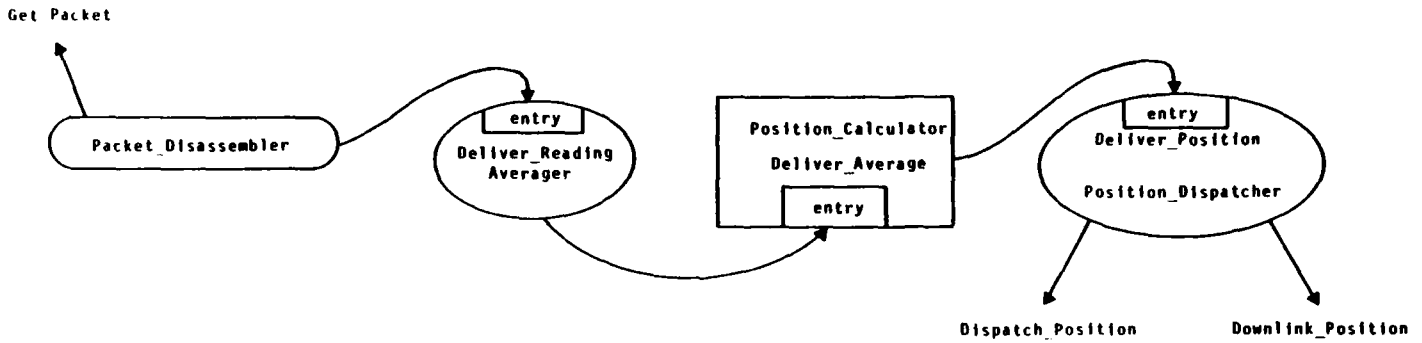
INSTRUCTOR NOTES

- WE HAVE SEEN THESE STREAM TRANSFORMATIONS TWICE BEFORE, SO THE CLASS SHOULD UNDERSTAND THEM.

- BULLET 4 - THIS WOULD ALMOST SURELY REQUIRE A PRAGMA.

TASK OPTIMIZATIONS 3: "INVERTING" STREAM TRANSFORMATIONS

- THE NAVIGATION PROBLEM (EXERCISE 15.1) HAS FOUR TASKS PERFORMING STREAM TRANSFORMATIONS.



- CAN VIEW
  - Position_Calculator AS AGENT FOR Averager
  - Averager AS AGENT FOR Packet_Disassembler

  WHICH ALLOWS Averager, Position_Calculator AND Position_Dispatcher TO EXECUTE AS COROUTINES
- ALTERNATELY, CAN RECOGNIZE STREAM TRANSFORMATIONS AS SPECIAL CASE.
  - ALL FOUR TASKS CAN EXECUTE AS COROUTINES
  - INVERSION MAY BE APPLIED (THIS WAS APPLIED MANUALLY IN SECTION 24)
- A COMPILER PROBABLY WILL NEED HELP DETECTING STREAMS - PRAGMA.

INSTRUCTOR NOTES

- COMMON SUBEXPRESSIONS OCCUR IN USER PROGRAMS AND IN CODE GENERATED BY A COMPILER.

- BULLET 1:

  IF THE VALUE OF ONE OF THE TERMS OF THE EXPRESSION CHANGES BETWEEN APPEARANCES OF THE EXPRESSION, THEN IT IS NOT A COMMON SUBEXPRESSION, E.G. IN THE SEQUENCE.

```
B := Y*Z
Y := Q
A := Y*Z
```

  THEN Y*Z IS NOT A COMMON SUBEXPRESSION. DO NOT BRING THIS UP UNLESS ASKED.

COMMON SUBEXPRESSIONS

- IF AN EXPRESSION APPEARS MORE THAN ONCE, THE COMPILER MAY BE ABLE TO EVALUATE IT ONCE AND USE THE RESULTING VALUE WHEREVER THE EXPRESSION APPEARS.

- EXAMPLE: USING THE COMMON SUBEXPRESSION A*B A COMPILER CAN TRANSLATE THE SEQUENCE OF STATEMENTS

```
W := A*B + C;
X := W + D;
Y := A*B + E;
Z := A*B + C;
```

AS IF IT HAD BEEN WRITTEN

```
Temp := A*B;
W := Temp + C;
X :=  W + D;
Y := Temp + E;
Z := W;
```

INSTRUCTOR NOTES

- WHEN THE VALUE OF AN EXPRESSION IS CONSTANT WITHIN A LOOP, EVALUATING THE EXPRESSION EACH TIME THROUGH THE LOOP IS WASTEFUL. HOWEVER, IT MAY BE CLEARER TO WRITE SUCH EXPRESSIONS WITHIN A LOOP. THIS OPTIMIZATION ALLOWS THE USER TO WRITE CODE WITHOUT WORRYING ABOUT REDUNDANT EXPRESSION EVALUATION.

CODE MOTION

- WHEN THE VALUE OF AN EXPRESSION IS CONSTANT WITHIN A LOOP, ITS EVALUATION CAN BE MOVED OUTSIDE OF THE LOOP.

- EXAMPLE: THE LOOP

```
for I in 1 .. Table'Last loop
   Table (I) := Table (I) * Factor**2;
end loop;
```

CAN BE COMPILED AS IF THE EXPRESSION Factor**2 WERE MOVED OUTSIDE OF THE LOOP:

```
Temp := Factor**2;
for I in 1 .. Table'Last loop
   Table (I) := Table (I)*Temp;
end loop;
```

INSTRUCTOR NOTES

- EXAMPLE 1:

  CATENATION IS MUCH MORE EXPENSIVE THAN ADDITION, SO THE SUM OF THE LENGTH OF BOTH STRINGS IS MUCH CHEAPER THAN THE LENGTH OF THE CATENATION OF BOTH PIECES.

- EXAMPLE 2:
  - IN THIS EXAMPLE, THE PRODUCT OF EVERY FOURTH ELEMENT OF A IS FORMED.
  - THE BOXES MARK THE THREE CHANGES THAT WERE MADE.
    - 4 IS ADDED TO I INSTEAD OF 1
    - I*4 IS REPLACED BY I, SINCE ADDING 4 TO I ACCOMPLISHES THE SAME THING
    - THE EXIT CONDITION IS CHANGED TO CHECK FOR I = 200 INSTEAD OF I = 50 (IN THE FIRST VERSION, I = 50 CORRESPONDED TO I*4 = 200).

  THE COSTLIER MULTIPLICATION WAS CHANGED TO A CHEAPER ADDITION.

STRENGTH REDUCTION

- EXPENSIVE OPERATIONS CAN SOMETIMES BE REPLACED BY CHEAPER ONES.
- EXAMPLE 1 THE CATENATION OPERATION IN

```
L := (S1 & S2)'Length;
```

CAN BE REPLACED BY CHEAPER ADDITION OPERATION IN

```
L := S1'Length + S2'Length;
```

- EXAMPLE 2: GIVEN THE ARRAY

```
A : array (1 .. 200) of Natural;
```

THE STATEMENTS

```
Product := 1;
I := 0;
loop
   I := I + 1;
   Product := Product * A(I*4);
   exit when I = 50;
end loop;
```

CAN BE COMPILED AS IF THEY HAD BEEN WRITTEN AS:

```
Product := 1;
[J] := 0;
loop
   [J] := [J] + [4];
   Product := Product * A ( [J] );
   exit when [J] = [200];
end loop;
```

(J TAKES ON THE SUCCESSIVE VALUES OF I*4.)

ON TYPICAL COMPUTERS, ADDITION IS FASTER THAN MULTIPLICATION.

INSTRUCTOR NOTES

- THIS EXAMPLE SHOULD ESPECIALLY APPEAL TO THE FORTRAN PROGRAMMER.
- THE THIRD for LOOP IS "JAMMED" INTO THE FIRST LOOP, ELIMINATING THE NEED FOR THE THIRD LOOP.

LOOP JAMMING

- WHEN TWO LOOPS ARE EXECUTED THE SAME NUMBER OF TIMES WITH THE SAME LOOP INDEX, THE BODIES CAN SOMETIMES BE MERGED.

- EXAMPLE: THE LOOPS

```
for I in 1 .. A'Last(1) loop
   for J in 1 .. A'Last(2) loop
      A (I, J) := 0;
   end loop;
end loop;

for I in 1 .. A'Last(1) loop
   A (I, I) := 1;
end loop;
```

CAN BE COMPILED AS IF THEY HAD BEEN WRITTEN AS:

```
for I in 1 .. A'Last(1) loop
   for J in 1 ..A'Last(2) loop
      A (I, J) := 0;
   end loop;
   A (I, I) := 1;
end loop;
```

INSTRUCTOR NOTES

- WE NEED SOMETHING LIKE AN IBM 370

  MVC From (Length), To

  INSTRUCTION

SPECIAL LOOP OPTIMIZATION

- GIVEN THE TWO DIMENSIONAL ARRAY A, WITH FLOAT COMPONENTS, THE LOOPS

```
for I in 1 .. N loop
   for J in 1 .. N loop
      A (I, J) := 0.0;
   end loop;
end loop;
```

CAN BE COMPILED AS IF WRITTEN AS

```
A := (A'Range (1) => (A'Range (2) => 0.0));
```

ON SOME MACHINES THIS CAN BE COMPILED EFFICIENTLY AS A "BLOCK MOVE."

INSTRUCTOR NOTES

- EXAMPLE 2 - THIS IS A GOOD WAY TO ACHIEVE CONDITIONAL COMPILATION.

DEAD CODE ELIMINATION

- WHEN A SECTION OF CODE CAN NEVER BE EXECUTED BECAUSE PROGRAM FLOW WILL NEVER REACH THE SECTION, THE CODE MAY BE DISCARDED BY THE COMPILER.

- EXAMPLE 1: THE STATEMENTS

```
A := 7;
if A = 0 then
   raise A_Is_Zero;
else
   C := B/A;
end if;
```

CAN BE REDUCED TO

```
A := 7
C := B/7;
```

- EXAMPLE 2: IN THE PREDEFINED PACKAGE System, THERE IS A ENUMERATION TYPE System.Name THAT NAMES THE SYSTEMS THAT THE COMPILER CAN GENERATE CODE FOR. THE CONSTANT System.System_Name IS A CONSTANT IN THIS TYPE. ASSUMING THAT Vax MC68000, Z8000 ARE ENUMERATION LITERALS IN THIS TYPE AND THAT System_Name IS MC68000, THE STATEMENT

```
case System.System_Name is
   when Vax =>
      Vax_Code_Procedure;
   when MC68000 =>
      MC68000_Code_Procedure;
   when Z8000 =>
      Z8000_Code_Procedure;
end case;
```

CAN BE REDUCED TO

```
MC68000_Code_Procedure;
```

INSTRUCTOR NOTES

LET THE COMPILER WORK FOR YOU

- "SMART" COMPILERS CAN PRODUCE HIGH QUALITY CODE.
- WRITE YOUR PROGRAM FOR CLARITY AND LET THE COMPILER GENERATE EFFICIENT OBJECT CODE.
- IF YOU FIND PERFORMANCE PROBLEMS THEN WORRY ABOUT TUNING.
- REMEMBER, THE COMPILER DOES NOT KNOW THAT A BINARY SEARCH IS FASTER THAN A LINEAR SEARCH.

INSTRUCTOR NOTES

PART VII

MODULE WRAP-UP

27. A COMPLETE EXAMPLE

INSTRUCTOR NOTES

THE MODULE FINISHES BY EXAMINING A COMPLETE REAL-TIME PROGRAM. THE PROGRAM IS MINUSCULE BY Ada STANDARDS: EXCEPT FOR THE HARDWARE INTERFACE PACKAGES AND THE USER INTERFACE (NOT SHOWN HERE) IT IS ONLY ABOUT 500 LINES LONG. NONETHELESS, IT ILLUSTRATES MANY OF THE MOST IMPORTANT IDEAS OF THE COURSE AND SHOWS HOW TASKS FIT INTO A COMPLETE PROGRAM.

THE PACE AND DEPTH WITH WHICH THIS SECTION IS TAUGHT CAN VARY, DEPENDING ON THE AMOUNT OF TIME LEFT. COMPLETE IN-DEPTH TREATMENT WILL REQUIRE FROM 1 1/4 TO 1 1/2 HOURS.

Section 27

A COMPLETE EXAMPLE

INSTRUCTOR NOTES

THE EXAMPLE WILL BE THE MULTIZONE HEATING SYSTEM THAT WE HAVE ALREADY LOOKED AT BRIEFLY IN SECTIONS 2 AND 11. THIS SLIDE GIVES INFORMAL REQUIREMENTS FOR THE SYSTEM. MAKE SURE EVERYONE UNDERSTANDS THE REQUIREMENTS BEFORE GOING ON.

- BULLET 3: THE ACTUAL TEMPERATURE CAN BECOME LOWER, OR MORE THAN 2 DEGREES HIGHER, THAN THE DESIRED TEMPERATURE EITHER BECAUSE THE ACTUAL TEMPERATURE CHANGES OR BECAUSE THE DESIRED TEMPERATURE CHANGES. (POINT THIS OUT ONLY AFTER COVERING BULLET 6.)

- BULLET 7: THE PROGRAM GIVES THE EXACT CORRESPONDENCE BETWEEN NUMBER OF OPEN VENTS AND HEATER SETTING. IF NO VENTS ARE OPEN, THE HEATER IS SET TOO LOW IF IT IS WARMING UP AND OFF OTHERWISE.

- BULLET 8: EMPHASIZE THAT IT IS NOT ACCEPTABLE IN MOST EMBEDDED APPLICATIONS TO IGNORE HARDWARE FAILURE. THIS IS JUST A SIMPLIFYING ASSUMPTION TO LIMIT THE COMPLEXITY OF THE EXAMPLE. (TYPICAL KINDS OF FAILURE ARE A STUCK VENT, A HEATER THAT WON'T TURN ON, OR A HEATER THAT OVERHEATS. A ROOM THAT DOES NOT BECOME WARM IN A REASONABLE AMOUNT OF TIME SHOULD PROBABLY ALSO TRIGGER AN ALARM.)

HEATING SYSTEM REQUIREMENTS

- HOT AIR HEATING SYSTEM WITH FIVE ZONES FED BY A CENTRAL HEATER.
- EACH ZONE HAS A THERMOMETER AND A VENT THAT CAN BE OPENED OR CLOSED UNDER COMPUTER CONTROL. THE TEMPERATURE MUST BE MONITORED FOUR TIMES A MINUTE.
- IF THE HEATER HAS ALREADY BEEN ON FOR TWO MINUTES, A ZONE'S VENT IS OPENED AS SOON AS ITS ACTUAL TEMPERATURE BECOMES LOWER THAN ITS DESIRED TEMPERATURE. THE VENT STAYS OPEN UNTIL THE ACTUAL TEMPERATURE IS AT LEAST TWO DEGREES HIGHER THAN THE DESIRED TEMPERATURE.
- IF THE HEATER HAS NOT BEEN ON FOR TWO MINUTES WHEN A ZONE REQUIRES HEAT, THE VENT FOR THAT ZONE MUST NOT BE OPENED UNTIL THE HEATER HAS WARMED UP FOR TWO MINUTES.
- THE DESIRED TEMPERATURE FOR A GIVEN ZONE VARIES ACCORDING TO THE TIME OF DAY, INDEPENDENTLY OF THE DESIRED TEMPERATURE FOR OTHER ZONES.
- AN OPERATOR SITTING AT A CONSOLE CONTROLS THE SCHEDULE OF CHANGES IN EACH ZONE'S DESIRED TEMPERATURE. THE OPERATOR HAS THREE COMMANDS AVAILABLE:
  - LIST THE SCHEDULED CHANGES IN DESIRED TEMPERATURE FOR A GIVEN ZONE.
  - ADD A REQUEST TO CHANGE THE DESIRED TEMPERATURE FOR ZONE *z* TO *d* DEGREES AT TIME *t*.
  - REMOVE THE SCHEDULED REQUEST TO CHANGE ZONE *z*'S DESIRED TEMPERATURE AT TIME *t*.

  CHANGES IN DESIRED TEMPERATURE MAY BE SCHEDULED FOR ANY MINUTE OF THE DAY AND MUST TAKE EFFECT WITHIN PLUS OR MINUS 30 SECONDS OF THE SCHEDULED TIME. THE OPERATOR MUST BE GIVEN APPROPRIATE ERROR MESSAGES WHEN HE ENTERS A SYNTACTICALLY INVALID COMMAND OR ATTEMPTS TO REMOVE A NONEXISTENT REQUEST.
- HEATER SETTING (OFF, LOW, MEDIUM, HIGH) ADJUSTED ACCORDING TO NUMBER OF OPEN VENTS.
- DON'T WORRY ABOUT HARDWARE FAILURE.

INSTRUCTOR NOTES

THIS IS THE FIRST OF TWO SLIDES DEPICTING THE DESIGN OF THE HEATING SYSTEM CONTROL PROGRAM. THIS SLIDE SHOWS THE TASK OBJECTS IN THE SYSTEM, THE FUNCTIONS PERFORMED BY EACH, AND THE ENTRIES CALLED BY EACH. THE NEXT SLIDE SHOWS WHERE TASKS ARE DECLARED AND WHICH PACKAGES AND SUBPROGRAMS THEY USE.

THE MAIN PROGRAM IS RESPONSIBLE FOR CONDUCTING A DIALOG WITH THE OPERATOR. WHEN A WELL-FORMED COMMAND IS ENTERED, AN APPROPRIATE ENTRY OF Scheduling_Task IS CALLED TO MANIPULATE THE SCHEDULE OF DESIRED TEMPERATURES. (Scheduling_Task SERVES AS A MONITOR FOR THE DATA STRUCTURE REPRESENTING THIS SCHEDULE.)

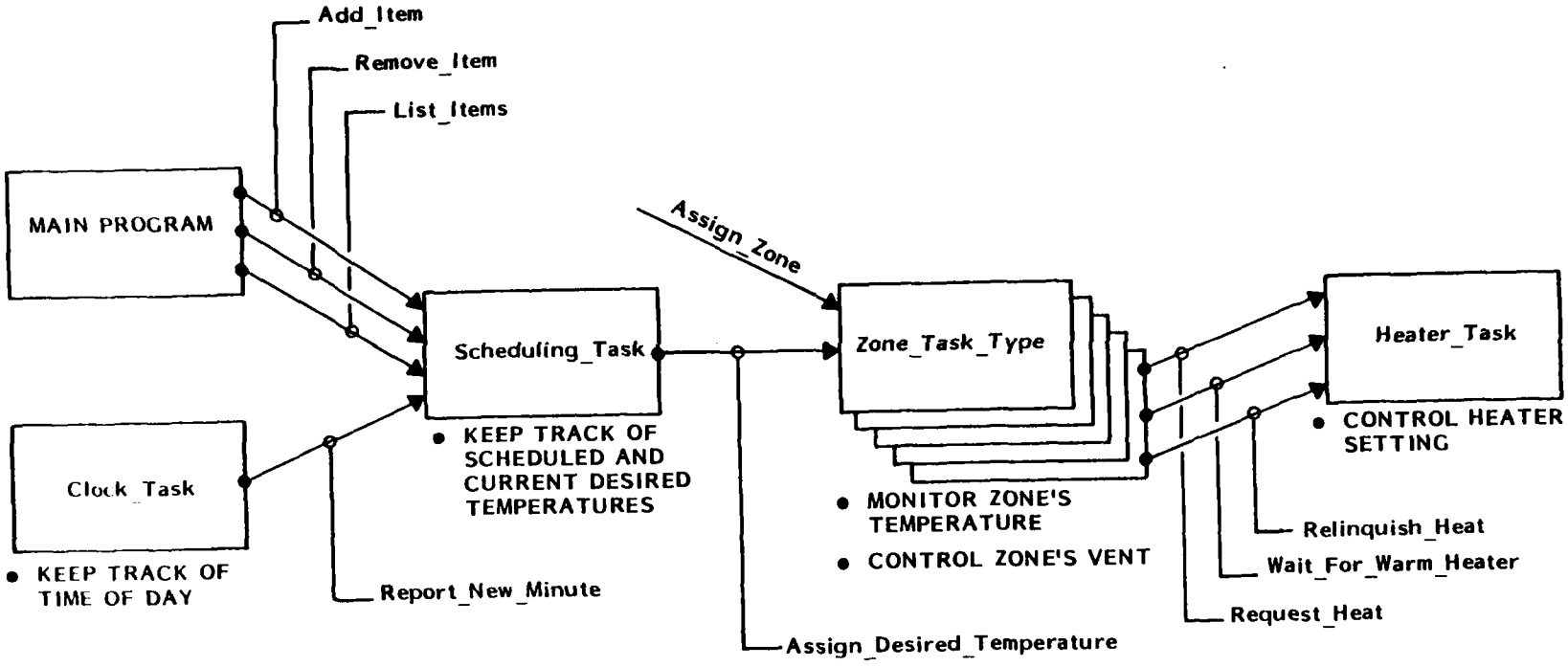
Clock_Task KEEPS TRACK OF THE TIME OF DAY AND CALLS Scheduling_Task.Report_New_Minute EACH TIME A NEW MINUTE BEGINS.

Scheduling_Task UPDATES THE SCHEDULE OF DESIRED TEMPERATURES WHEN Add_Item OR Remove_Item IS CALLED, DISPLAYS THE SCHEDULED TEMPERATURE CHANGES FOR A PARTICULAR ZONE WHEN List_Items IS CALLED, AND CHECKS WHETHER ANY CHANGES ARE SCHEDULED FOR THE CURRENT MINUTE WHEN Report_New_Minute IS CALLED. IF THERE IS A CHANGE SCHEDULED FOR THE CURRENT MINUTE, THE Assign_Desired_Temperature ENTRY OF THE APPROPRIATE Zone_Task_Type OBJECT IS CALLED.

THERE IS ONE Zone_Task_Type TASK OBJECT FOR EACH ZONE. THE Assign_Zone ENTRY IS CALLED ONCE DURING INITIALIZATION OF A LIBRARY PACKAGE, SO TELL EACH TASK WHICH THERMOMETER TO READ AND WHICH VENT TO CONTROL. THE TASK MONITORS THE TEMPERATURE ONCE EVERY 15 SECONDS, COMPARES IT WITH THE DESIRED TEMPERATURE, AND MANIPULATES THE VENTS ACCORDINGLY. THE DESIRED TEMPERATURE IS UPDATED WHENEVER A CALL ON Assign_Desired_Temperature ARRIVES. THE TASK CALLS FIRST Header_Task.Request_Heat AND THEN Heater_Task.Wait_For_Warm_Heater BEFORE OPENING A VENT, AND Heater_Task.Relinquish_Heat UPON CLOSING A VENT. THE CALL ON Request_Heat CAUSES THE HEATER TO BE STARTED IF IT WAS OFF: A CALL ON Wait_For_Warm_Heater IS NOT ACCEPTED UNTIL THE HEATER HAS BEEN ON FOR TWO MINUTES. (IF THE HEATER HAS BEEN ON FOR TWO MINUTES, BOTH THESE CALLS ARE ACCEPTED IMMEDIATELY.)

Heater_Task KEEPS TRACK OF THE NUMBER OF OPEN VENTS, AND MODIFIES THE HEATER SETTING ACCORDINGLY. IT ACCEPTS CALLS ON Wait_For_Warm_Heater ONLY WHEN THE HEATER HAS BEEN ON FOR AT LEAST TWO MINUTES.

HEATING SYSTEM TASKING STRUCTURE



INSTRUCTOR NOTES

THIS SLIDE DEPICTS THE PROGRAM UNITS IN THE HEATING SYSTEM CONTROL PROGRAM. WE WILL LOOK AT THE TEXT OF EACH NONSHADED PROGRAM UNIT.

THE PACKAGE Global_Data_Package (AT THE BOTTOM OF THE DIAGRAM) CONTAINS TYPE AND SUBTYPE DECLARATIONS USED THROUGHOUT THE PROGRAM.

THE PROCEDURE Control_Heating_System IS THE MAIN PROGRAM. THE NESTED PROCEDURE Get_Well_Formed_Command IS RESPONSIBLE FOR MOST OF THE INTERACTION WITH THE USER. EACH CALL ON Get_Well_Formed_Command OBTAINS ONE SYNTACTICALLY CORRECT REQUEST.

Scheduling_Package PROVIDES SCHEDULE-MANIPULATION SUBPROGRAMS THAT ARE CALLED BY THE MAIN PROGRAM. INTERNALLY, THESE SUBPROGRAMS CALL ENTRIES OF Scheduling_Task, WHICH IS HIDDEN INSIDE THE PACKAGE BODY. Clock_Task, WHICH ALSO CALLS AN ENTRY OF Scheduling_Task, IS ALSO HIDDEN IN THE Scheduling_Package BODY. THE NESTED PACKAGE Table_Package HIDES THE ACTUAL TABLE CONTAINING THE SCHEDULE OF TEMPERATURE CHANGES AND PROVIDES SUBPROGRAMS TO MANIPULATE THE TABLE. THE Scheduling_Task MONITOR IS IMPLEMENTED USING THE FACILITIES OF Table_Package.

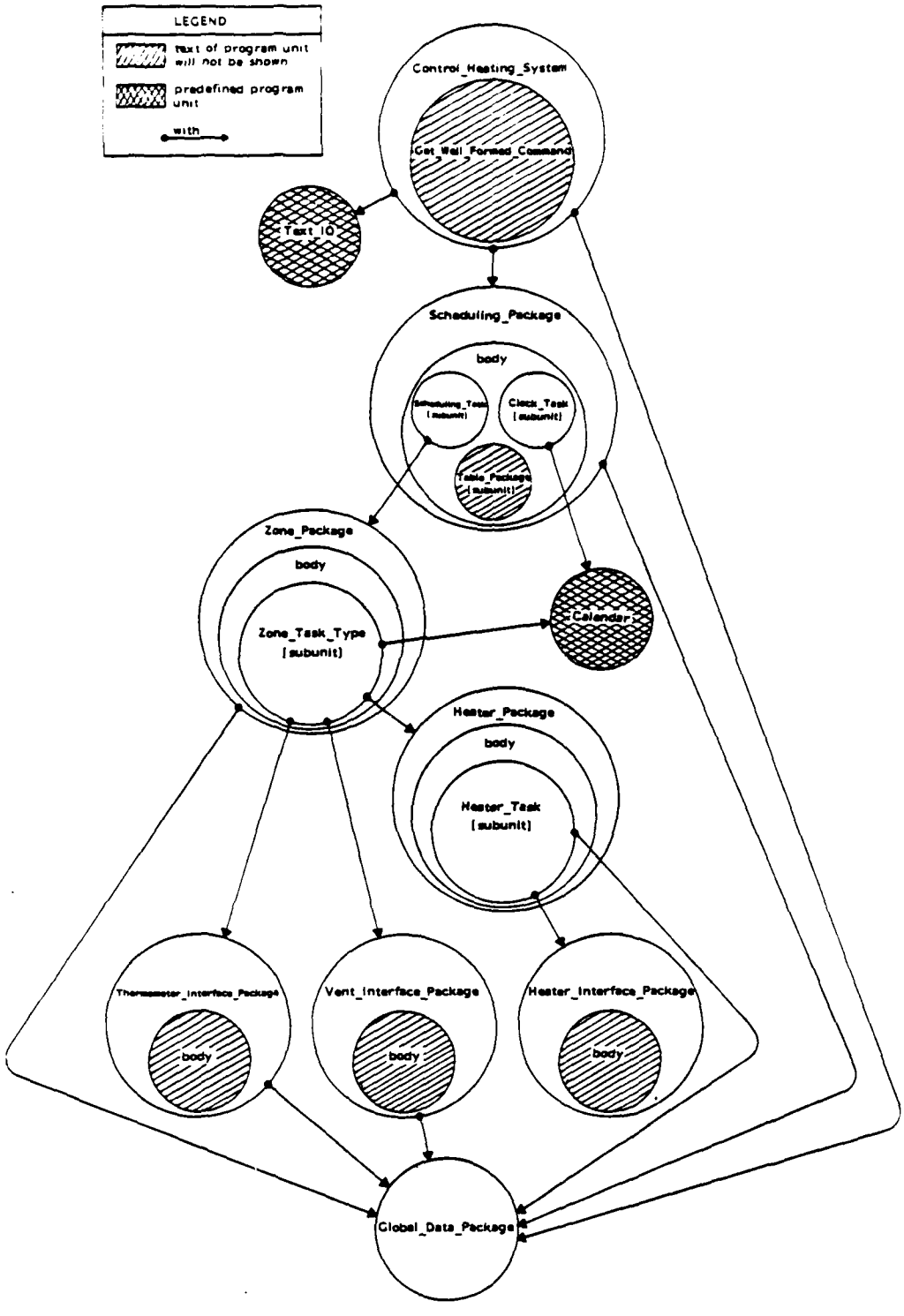
Zone_Package PROVIDES A SUBPROGRAM CALLED BY Scheduling_Task TO CHANGE THE CURRENT DESIRED TEMPERATURE FOR A SPECIFIED ZONE. Zone_Task_Type AND FIVE OBJECTS OF THAT TYPE ARE HIDDEN INSIDE THE Zone_Package BODY.

Thermometer_Interface_Package AND Vent_Interface_Package PROVIDE HARDWARE SUBPROGRAMS CALLED FROM THE Zone_Task_Type TASK BODY.

Heater_Package PROVIDES TWO SUBPROGRAMS CALLED FROM THE Zone_Task_Type TASK BODY, ONE JUST BEFORE A VENT IS OPENED AND ONE WHEN THE VENT IS CLOSED. THE FIRST SUBPROGRAM CALLS Heater_Task.Request_Heat AND THEN Heater_Task.Wait_For_Warm_Heater. THE SECOND CALLS Heater_Task.Relinquish_Heat. Heater_Task IS HIDDEN IN THE Heater_Package BODY.

Heater_Interface_Package PROVIDES A SUBPROGRAM CALLED BY Heater_Task FOR CONTROLLING THE HEATER SETTING.

HEATING SYSTEM PROGRAM UNIT STRUCTURE



INSTRUCTOR NOTES

THIS PACKAGE PROVIDES DEFINITIONS FOR Zone_Number_Subtype, Degrees_Type, AND Minute_Number_Subtype.

INTERNALLY, THE PROGRAM REPRESENTS TIMES OF DAY AS VALUES IN Minute_Number_Subtype. THESE ARE INTEGERS REPRESENTING THE NUMBER OF MINUTES SINCE MIDNIGHT.

DECLARATION OF GLOBAL DATA PACKAGE

```
package Global_Data_Package is

   subtype Zone_Number_Subtype is Integer range 1 .. 5;
   type Degrees_Type is delta 0.125 range 0.0 .. 128.0;
   subtype Minute_Number_Subtype is Integer range 0 .. 24*60 - 1;

end Global_Data_Package;

-- This package has no body.
```

INSTRUCTOR NOTES

THIS PACKAGE PROVIDES THE SUBPROGRAMS CALLED BY THE MAIN PROGRAM TO MANIPULATE THE *SCHEDULE OF TEMPERATURE CHANGES.*

Removal_Error IS RAISED BY Remove_Scheduled_Temperature IF NO TEMPERATURE CHANGE WAS SCHEDULED FOR THE DESIRED ZONE AT THE DESIRED TIME.

WE SHALL LOOK AT THE *PACKAGE BODY LATER.*

DECLARATION OF Scheduling_Package

```
with Global_Data_Package;

package Scheduling_Package is

   subtype Zone_Number_Subtype   is Global_Data_Package.Zone_Number_Subtype;
   subtype Degrees_Type          is Global_Data_Package.Degrees_Type;
   subtype Minute_Number_Subtype is Global_Data_Package.Minute_Number_Subtype;

   procedure Schedule_Temperature
      (Zone                : in Zone_Number_Subtype;
       Starting_Time       : in Minute_Number_Subtype;
       Desired_Temperature : in Degrees_Type);

   procedure Remove_Scheduled_Temperature
      (Zone          : in Zone_Number_Subtype;
       Starting_Time : in Minute_Number_Subtype);
       -- May raise Removal_Error

   procedure List_Schedule (Zone: in Zone_Number_Subtype);

   Removal_Error: exception;

end Scheduling_Package;
```

INSTRUCTOR NOTES

NOW THAT WE HAVE SEEN THE LIBRARY UNITS USED BY THE MAIN PROGRAM, WE CAN LOOK AT THE MAIN PROGRAM ITSELF. THE DECLARATIVE PART IS SHOWN ON THIS SLIDE AND THE SEQUENCE OF STATEMENTS IS SHOWN ON THE NEXT SLIDE.

THE DECLARATIVE PART CONTAINS AN ENUMERATION TYPE FOR THE THREE KINDS OF COMMANDS AND A RECORD TYPE WITH VARIANTS FOR COMMANDS THEMSELVES. A Command_Type VALUE HAS ONE OF THE FOLLOWING THREE FORMS:

- Command_Kind_Part (= Add_Command), Zone_Part, Time_Part, Temperature_Part
- Command_Kind_Part (= Remove_Command), Zone_Part, Time_Part
- Command_Kind_Part (= List_Command), Zone_Part

(TRY NOT TO GET BOGGED DOWN IN A DISCUSSION OF THE NESTED VARIANTS.)

THE DECLARATIVE PART CONTAINS THE STUB OF PROCEDURE Get_Well_Formed_Command, WHICH INTERACTS WITH THE OPERATOR, INTERPRETS INPUT, AND PROVIDES A Command_Type VALUE.

MAIN PROGRAM (SLIDE 1 of 2)

```
with Scheduling_Package, Global_Data_Package;
with Text_IO; use Text_IO;

procedure Control_Heating_System is

   type Command_Kind_Type is (Add_Command, Remove_Command, List_Command);

   type Command_Type (Command_Kind_Part: Command_Kind_Type := Add_Command) is
      record
         Zone_Part: Zone_Number_Subtype;
         case Command_Kind_Part is
            when Add_Command | Remove_Command =>
               Time_Part: Minute_Number_Subtype;
               case Command_Kind_Part is
                  when Add_Command =>
                     Temperature_Part: Degrees_Type;
                  when others =>
                     null;
               end case;
            when List_Command =>
               null;
         end case;
      end record;

   procedure Get_Well_Formed_Command (Command: out Command_Type) is separate;
      -- Will not be shown.

   Command: Command_Type;

begin

      (STATEMENTS ON NEXT SLIDE)

end Control_Heating_System;
```

INSTRUCTOR NOTES

THE PROCEDURE SIMPLY EXECUTES AN INFINITE LOOP THAT GETS A COMMAND AND PROCESSES IT BY CALLING THE APPROPRIATE Scheduling_Package PROCEDURE.

THE CALL ON Remove_Scheduled_Temperature OCCURS WITHIN A BLOCK STATEMENT THAT CONTAINS A HANDLER FOR Removal_Error.

MAIN PROGRAM (SLIDE 2 OF 2)

```
with Scheduling_Package, Global_Data_Package;
with Text_IO; use Text_IO;

procedure Control_Heating_System is

    (DECLARATIONS ON PREVIOUS SLIDE)
begin

   loop

      Get_Well_Formed_Command (Command);
      case Command.Command_Kind_Part is
         when Add_Command =>
            Scheduling_Package.Schedule_Temperature
               (Command.Zone_Part,
                Command.Time_Part,
                Command.Temperature_Part);
         when Remove_Command =>
            begin
               Scheduling_Package.Remove_Scheduled_Temperature
                  (Command.Zone_Part, Command.Time_Part);
            exception
               when Scheduling_Package.Removal_Error =>
                  Put_Line
                     ("*****ERROR: No temperature change scheduled" &
                         " for that zone and time.");
                  Put_Line ("*****COMMAND IGNORED.");
            end;
         when List_Command =>
            Scheduling_Package.List_Schedule (Command.Zone_Part);
      end case ;

   end loop;

end Control_Heating_System;
```

INSTRUCTOR NOTES

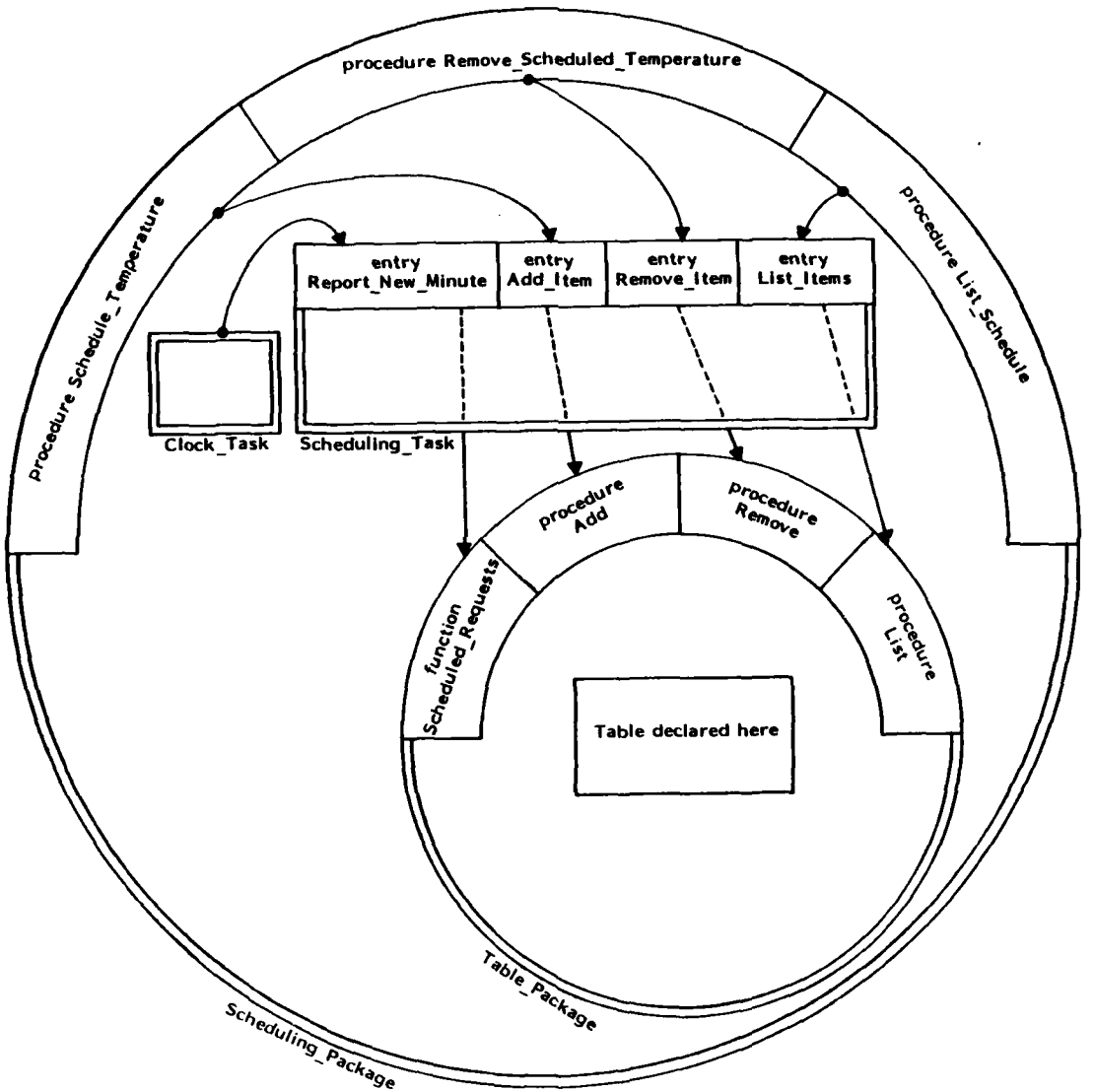
THIS SLIDE DEPICTS THE IMPLEMENTATION OF Scheduling_Package, WHOSE DECLARATION WAS SHOWN ON AN EARLIER SLIDE. THREE PROGRAM UNITS ARE DECLARED INSIDE THE Scheduling_Package BODY -- Clock_Task, Scheduling_Task, AND Table_Package. THE DATA STRUCTURE STORING SCHEDULED TEMPERATURE CHANGES IS DECLARED INSIDE THE Table_Package BODY AND MANIPULATED IN TERMS OF THE FOUR SUBPROGRAMS PROVIDED BY Table_Package. THE FUNCTION Scheduled_Requests RETURNS A LIST OF THOSE TEMPERATURE CHANGES SCHEDULED FOR A SPECIFIED MINUTE. (EACH ELEMENT OF THE LIST CONSISTS OF A ZONE AND A NEW TEMPERATURE.) THE PROCEDURES Add AND Remove MODIFY THE TABLE AND THE PROCEDURE LIST DISPLAYS THE SCHEDULED TEMPERATURES FOR A GIVEN ZONE ON THE OPERATOR CONSOLE.

CALLS ON Scheduled_Requests OCCUR EVERY MINUTE AND CALLS ON Add AND Remove OCCUR WHEN THE OPERATOR ENTERS THE CORRESPONDING COMMANDS. Scheduling_Task ACTS AS A MONITOR TO PREVENT ARBITRARY INTERLEAVING OF CALLS ON THESE SUBPROGRAMS.

Scheduling_Task PROVIDES OPERATIONS ON A SCHEDULE SHARED BY TWO TASKS, AND IMPLEMENTS THOSE OPERATIONS DIRECTLY IN TERMS OF Table_Package.

THE Report_New_Minute ENTRY IS CALLED EVERY MINUTE BY Clock_Task. SINCE THAT ENTRY AND THAT TASK ARE NOT PART OF THE INTERFACE NEEDED BY THE MAIN PROGRAM, WE HIDE Clock_Task AND Scheduling_Task INSIDE THE PACKAGE Scheduling_Package. THOSE SERVICES WHICH ARE RELEVANT TO THE MAIN PROGRAM ARE PROVIDED BY THE SUBPROGRAMS Schedule_Temperature, Remove_Scheduled_Temperature, AND List_Schedule. THESE SUBPROGRAMS ARE IMPLEMENTED DIRECTLY BY CALLS ON THE CORRESPONDING ENTRIES OF Scheduling_Task.

IMPLEMENTATION OF Scheduling_Package



INSTRUCTOR NOTES

THIS IS THE STRUCTURE OF THE PACKAGE BODY IMPLEMENTING THE DIAGRAM ON THE PREVIOUS SLIDE. THE FOLLOWING SLIDES SHOW THE TASK DECLARATIONS, THE Table_Package DECLARATION, AND MOST OF THE BODY STUBS.

SKELETON OF Scheduling_Package BODY

```
package body Scheduling_Package is

    (TASK DECLARATIONS)

    (Table_Package DECLARATION)

   procedure Schedule_Temperature
      (Zone                : in Zone_Number_Subtype;
       Starting_Time       : in Minute_Number_Subtype;
       Desired_Temperature : in Degrees_Type)
      is separate;

   procedure Remove_Scheduled_Temperature
      (Zone          : in Zone_Number_Subtype;
       Starting_Time : in Minute_Number_Subtype)
      is separate;

   procedure List_Schedule (Zone: in Zone_Number_Subtype) is separate;

   task body Scheduling_Task is separate;

   task body Clock_Task is separate;

   package body Table_Package is separate;  -- Subunit not shown here

end Scheduling_Package;
```

INSTRUCTOR NOTES

THIS SLIDE SHOWS THE DECLARATIONS OF Scheduling_Task WITH FOUR ENTRIES AND Clock_Task WITH NO ENTRIES. THE ONLY FUNCTION OF Clock_Task IS TO CALL Scheduling_Task.Report_New_Minute EVERY MINUTE WITH THE CURRENT MINUTE NUMBER.

POINT OUT THE COMMENT FOLLOWING THE DECLARATION OF Remove_Item. Removal_Error WAS DECLARED IN THE Scheduling_Package DECLARATION SHOWN EARLIER.



TASK DECLARATIONS INSIDE Scheduling_Package BODY

```
package body Scheduling_Package is

   task Scheduling_Task is

      entry Report_New_Minute (Current_Minute: in Minute_Number_Subtype);

      entry Add_Item
         (Zone                : in Zone_Number_Subtype;
          Starting_Time       : in Minute_Number_Subtype;
          Desired_Temperature : in Degrees_Type);

      entry Remove_Item
         (Zone          : in Zone_Number_Subtype;
          Starting_Time : in Minute_Number_Subtype);
         -- Entry call may raise Removal_Error.

      entry List_Items (Zone: in Zone_Number_Subtype);

   end Scheduling_Task ;

   task Clock_Task;

   (Table_Package DECLARATION)

   (BODY STUBS)

end Scheduling_Package;
```

INSTRUCTOR NOTES

THIS IS THE DECLARATION OF THE PACKAGE PROVIDING A SCHEDULE AND NONCONCURRENT OPERATIONS ON THE SCHEDULE.

THE VALUE RETURNED BY Scheduled_Requests IS A POINTER TO A Request_List_Type ARRAY. THIS ARRAY REPRESENTS A LIST OF TEMPERATURE CHANGES SCHEDULED FOR THE MINUTE SPECIFIED AS THE FUNCTION PARAMETER. EACH ELEMENT OF THE LIST IS A Change_Request_Type RECORD SPECIFYING A ZONE AND A NEW TEMPERATURE FOR THAT ZONE. SINCE Request_List_Type IS UNCONSTRAINED, THE SIZE OF THE ARRAY POINTED TO BY THE FUNCTION RESULT CAN VARY. FOR MOST MINUTES, NO TEMPERATURE CHANGES ARE SCHEDULED, SO THE FUNCTION RESULT POINTS TO A NULL ARRAY (REPRESENTING THE EMPTY LIST).

POINT OUT THE COMMENT FOLLOWING THE DECLARATION OF Remove.

THE SUBUNIT FOR THE Table_Package BODY IS NOT SHOWN HERE. SINCE THERE IS ONLY 1440 Minute_Number_Subtype VALUES, A REASONABLE REPRESENTATION OF THE SCHEDULE IS AS AN

```
array (Minute_Number_Subtype) of Request_List_Pointer_Type
```

PACKAGE DECLARATION INSIDE Scheduling_Package BODY

```
package body Scheduling_Package is
   (TASK DECLARATIONS)
   package Table_Package is
      type Change_Request_Type is
         record
            Zone_Part            : Zone_Number_Subtype;
            New_Temperature_Part : Degrees_Type;
         end record;
      type Request_List_Type is
         array (Positive range <>) of Change_Request_Type;
      type Request_List_Pointer_Type is access Request_List_Type;
      function Scheduled_Requests
         (Minute: Minute_Number_Subtype) return Request_List_Pointer_Type;
      procedure Add
         (Zone                : in Zone_Number_Subtype;
          Starting_Time       : in Minute_Number_Subtype;
          Desired_Temperature : in Degrees_Type);
      procedure Remove
         (Zone          : in Zone_Number_Subtype;
          Starting_Time : in Minute_Number_Subtype);
         -- May raise Removal_Error.
      procedure List (Zone: in Zone_Number_Subtype);
   end Table_Package;
   (BODY STUBS)
end Scheduling_Package;
```

INSTRUCTOR NOTES

THESE ARE SUBUNITS OF THE PROCEDURES PROVIDED BY Scheduling_Package. EACH IS IMPLEMENTED DIRECTLY BY CALLING ONE OF THE ENTRIES OF Scheduling_Task.

THE HANDLER FOR Removal_Error IN Remove_Scheduled_Temperature IS NOT ACTUALLY NECESSARY. WITHOUT IT, THE EXCEPTION WOULD BE PROPAGATED IMPLICITLY. THE EXPLICIT PROPAGATION SERVES TWO PURPOSES:

- TO DOCUMENT THE FACT THAT Remove_Scheduled_Temperature SOMETIMES RAISES Removal_Error, AND MAKE IT EASIER FOR SOMEONE READING THE PROGRAM TO TRACK THE USE OF THIS EXCEPTION.

- TO GUARD AGAINST THE POSSIBILITY THAT SOMEONE MIGHT LATER ADD A "when other" HANDLER TO HANDLE ALL UNANTICIPATED EXCEPTIONS. WITHOUT AN EXPLICIT HANDLER FOR Removal_Error, THE "when others" HANDLER WOULD ALSO HANDLE Removal_Error, AND IT WOULD NOT BE PROPAGATED.

SUBUNITS FOR PROCEDURES PROVIDED BY Scheduling_Package

```
separate (Scheduling_Package)

procedure Schedule_Temperature
   (Zone                : in Zone_Number_Subtype;
    Starting_Time       : in Minute_Number_Subtype;
    Desired_Temperature : in Degrees_Type) is
begin
   Scheduling_Task.Add_Item (Zone, Starting_Time, Desired_Temperature);
end Schedule_Temperature;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

separate (Scheduling_Package)

procedure Remove_Scheduled_Temperature
   (Zone          : in Zone_Number_Subtype;
    Starting_Time : in Minute_Number_Subtype) is
begin
   Scheduling_Task.Remove_Item (Zone, Starting_Time);
end Remove_Scheduled_Temperature;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

separate (Scheduling_Package)

procedure List_Schedule (Zone: in Zone_Number_Subtype) is
begin
   Scheduling_Task.List_Items (Zone);
end List_Schedule;
```

INSTRUCTOR NOTES

THIS IS THE BODY OF Clock_Task, WHICH CALLS Scheduling_Task.Report_New_Minute ONCE EVERY 60 SECONDS. USE OF Next_Tick_Time AVOIDS CUMULATIVE DRIFT.

THE PREDEFINED FUNCTION Calendar.Seconds RETURNS THE NUMBER OF SECONDS SINCE MIDNIGHT. DIVIDING BY 60 YIELDS A Duration VALUE GIVING THE NUMBER OF MINUTES SINCE MIDNIGHT. (SINCE Duration IS A FIXED-POINT TYPE, DIVISION BY AN Integer VALUE IS PERMITTED.) CONVERTING THIS REAL VALUE TO THE INTEGER SUBTYPE Minute_Number_Subtype ROUNDS TO THE NEAREST WHOLE MINUTE NUMBER. THIS SATISFIES THE REQUIREMENT THAT TEMPERATURE CHANGES OCCUR WITHIN PLUS OR MINUS 30 SECONDS OF THE SCHEDULED MINUTE. (A MORE EXACT SOLUTION WOULD INITIALIZE Next_Tick_Time TO THE NEXT TIME FOR WHICH Calendar.Seconds WOULD RETURN AN EXACT MULTIPLE OF 60. THEN DELAYS WOULD BE SCHEDULED TO EXPIRE AT WHOLE NUMBERS OF SECONDS AFTER MIDNIGHT.)

SUBUNIT FOR Clock_Task

```
with Calendar;

separate (Scheduling_Package)

task body Clock_Task is

   Startup_Time, Next_Tick_Time : Calendar.Time;
   Minute_Number                : Minute_Number_Subtype;

begin

   Startup_Time := Calendar.Clock;
   Minute_Number :=
      Minute_Number_Subtype (Calendar.Seconds (Startup_Time) / 60) ;
   Next_Tick_Time := Startup_Time + 60.0;

   loop
      delay Next_Tick_Time - Calendar.Clock;
      Next_Tick_Time := Next_Tick_Time + 60.0;
      if Minute_Number = Minute_Number_Subtype'Last then
         Minute_Number := 0;
      else
         Minute_Number := Minute_Number + 1;
      end if;
      Scheduling_Task.Report_New_Minute (Minute_Number);
   end loop;

end Clock_Task;
```

INSTRUCTOR NOTES

SINCE THE Scheduling_Task BODY CALLS A PROCEDURE PROVIDED BY Zone_Package, WE'LL LOOK AT THE Zone_Package DECLARATION BEFORE EXAMINING THE Scheduling_Task SUBUNIT.

Update_Desired_Temperature IS CALLED TO CHANGE THE CURRENT DESIRED TEMPERATURE OF A SPECIFIED ZONE. WE'LL LOOK AT THE IMPLEMENTATION LATER.

DECLARATION OF Zone_Package

```
with Global_Data_Package;

package Zone_Package is

   subtype Zone_Number_Subtype is Global_Data_Package.Zone_Number_Subtype;
   subtype Degrees_Type        is Global_Data_Package.Degrees_Type;

   procedure Update_Desired_Temperature
      (Zone        : in Zone_Number_Subtype;
       Temperature : in Degrees_Type);

end Zone_Package;
```

INSTRUCTOR NOTES

THE HEART OF THE TASK IS THE LOOP, WHICH IS SHOWN ON THE NEXT SLIDE. THIS SLIDE SIMPLY ESTABLISHES THE CONTEXT FOR THIS LOOP.

SUBUNIT FOR Scheduling_Task (SLIDE 1 OF 2)

```
with Zone_Package;

separate (Scheduling_Package)

task body Scheduling_Task is

   Minute_Number : Minute_Number_Subtype;
   Change_List   : Table_Package.Request_List_Pointer_Type;

begin

   loop

       (LOOP SHOWN ON NEXT SLIDE)

   end loop;

end Scheduling_Task;
```

INSTRUCTOR NOTES

THE LOOP CONTAINS A SELECTIVE WAIT WITH AN ALTERNATIVE FOR EACH OF THE Scheduling_Task ENTRIES. EACH OF THESE ALTERNATIVES INVOKES A SUBPROGRAM FROM Table_Package TO DO ITS WORK.

IN THE FIRST accept ALTERNATIVE, Table_Package.Scheduled_Requests RETURNS A POINTER TO A LIST OF Table_Package.Change_Request_Type RECORDS. (SEE SLIDE 27-11.) Zone_Package.Update_Desired_Temperature IS CALLED ONCE FOR EACH RECORD IN THE ARRAY, WITH THE RECORD COMPONENTS USED AS PARAMETERS. (Change_List'Range DENOTES THE INDEX RANGE OF THE ARRAY POINTED TO BY Change_List, AND Change_List (C) DENOTES A COMPONENT OF THE ARRAY POINTED TO BY Change_List. TYPICALLY, Change_List POINTS TO A NULL ARRAY, SO Change_List'Range IS A NULL RANGE AND THE for LOOP IS EXECUTED ZERO TIMES.)

THE OTHER accept ALTERNATIVES SIMPLY CONTAIN CALLS ON THE CORRESPONDING Table_Package PROCEDURES. IN THE THIRD ALTERNATIVE, THE CALL ON Table_Package.Remove MAY RAISE Removal_Error. SINCE THE CALL IS INSIDE THE accept STATEMENT, THE EXCEPTION IS PROPAGATED BOTH TO THE ENTRY CALL AND TO THE Scheduling_Task BODY. THE SELECTIVE WAIT IS ENCLOSED IN A BLOCK STATEMENT WITH A NULL HANDLER SO THAT THE EXCEPTION RAISED IN THE TASK BODY CAN BE IGNORED.

SUBUNIT FOR Scheduling_Task (SLIDE 2 OF 2)

```
loop
  begin
    select
      accept Report_New_Minute (Current_Minute: in Minute_Number_Subtype) do
        Minute_Number := Current_Minute;
      end Report_New_Minute;
      Change_List := Table_Package.Scheduled_Requests (Minute_Number);
      for C in Change_List'Range loop
        Zone_Package.Update_Desired_Temperature
          (Change_List (C).Zone_Part, Change_List (C).New_Temperature_Part);
      end loop;
    or
      accept Add_Item
        (Zone                : in Zone_Number_Subtype;
         Starting_Time       : in Minute_Number_Subtype;
         Desired_Temperature : in Degrees_Type) do
         Table_Package.Add (Zone, Starting_Time, Desired_Temperature);
      end Add_Item;
    or
      accept Remove_Item
        (Zone          : in Zone_Number_Subtype;
         Starting_Time : in Minute_Number_Subtype) do
        Table_Package.Remove (Zone, Starting_Time);
          -- Removal_Error may be raised here.
      end Remove_Item;
    or
     accept List_Items (Zone: in Zone_Number_Subtype) do
        Table_Package.List (Zone);
      end List_Items;
    end select;
  exception
    when Removal_Error => null;
  end;
end loop;
```

INSTRUCTOR NOTES

THIS IS THE BODY OF THE PACKAGE DECLARED TWO SLIDES EARLIER. IT CONTAINS A DECLARATION OF THE TASK TYPE Zone_Task_Type AND AN ARRAY Zone_Task_List CONTAINING ONE TASK OF THIS TYPE FOR EACH ZONE. Zone_Task_Type HAS AN INITIALIZATION ENTRY Assign_Zone TO INFORM EACH TASK OBJECT OF THE ZONE FOR WHICH IT IS RESPONSIBLE. THERE IS ALSO AN Assign_Desired_Temperature ENTRY CALLED TO CHANGE THE DESIRED TEMPERATURE FOR THE CORRESPONDING ZONE.

THE PROCEDURE Update_Desired_Temperature PROVIDED BY Zone_Package SIMPLY CALLS THE Assign_Designed_Temperature ENTRY OF THE APPROPRIATE TASK OBJECT. THE FIRST PARAMETER OF Update_Desired_Temperature IS USED AS AN ARRAY INDEX INTO Zone_Task_List AND THE SECOND IS PASSED ALONG AS A PARAMETER TO THE ENTRY CALL.

BECAUSE Zone_Package IS A LIBRARY UNIT, ELABORATION OF THE PACKAGE BODY TAKES PLACE BEFORE EXECUTION OF THE MAIN PROGRAM. THE COMPONENTS OF Zone_Task_List ARE ACTIVATED JUST BEFORE THE SEQUENCE OF STATEMENTS IN THE PACKAGE BODY IS EXECUTED. THAT SEQUENCE OF STATEMENTS CALLS THE Assign_Zone ENTRY OF EACH COMPONENT OF Zone_Task_List. EACH COMPONENT IS PASSED ITS INDEX WITHIN THE ARRAY AS AN ENTRY PARAMETER. THIS TAKES PLACE BEFORE ANY CALLS ON Update_Desired_Temperature.

WE'LL LOOK AT THE Zone_Task_Type TASK BODY LATER.

BODY OF Zone_Package

```
package body Zone_Package is

   task type Zone_Task_Type is
      entry Assign_Zone (Zone: in Zone_Number_Subtype);
      entry Assign_Desired_Temperature (Temperature: in Temperature_Type);
   end Zone_Task_Type;

   task body Zone_Task_Type is separate;

   Zone_Task_List: array (Zone_Number_Subtype) of Zone_Task_Type;

   procedure Update_Desired_Temperature
      (Zone        : in Zone_Number_Subtype;
       Temperature : in Degrees_Type) is
   begin
      Zone_Task_List (Zone).Assign_Desired_Temperature (Temperature);
   end Update_Desired_Temperature;

begin  -- Zone_Package initialization

   for Zone in Zone_Number_Subtype loop
      Zone_Task_List (Zone).Assign_Zone (Zone);
   end loop;

end Zone_Package;
```

INSTRUCTOR NOTES

BEFORE EXAMINING THE Zone_Task SUBUNIT, WE LOOK AT THREE PACKAGES USED BY THE TASK BODY.

Thermometer_Interface_Package PROVIDES A FUNCTION FOR DETERMINING THE CURRENT TEMPERATURE IN A SPECIFIED ZONE.

Vent_Interface_Package PROVIDES PROCEDURES FOR OPENING AND CLOSING THE VENT IN A SPECIFIED ZONE.

Heater_Package PROVIDES TWO PROCEDURES FOR COORDINATING VENT OPENINGS AND CLOSINGS WITH HEATER SETTINGS. Get_Open_Vent_CLearance SHOULD BE CALLED BEFORE OPENING A VENT. THIS CALL WILL NOT COMPLETE BEFORE THE HEATER HAS BEEN ON FOR AT LEAST TWO MINUTES. Report_Vent_Closed SHOULD BE CALLED WHEN A VENT IS CLOSED.

WE WILL NOT SHOW THE BODIES OF Thermometer_Interface_Package AND Vent_Interface_Package (WHICH CONSIST LARGELY OF LOW LEVEL OPERATIONS). WE'LL LOOK AT THE Heater_Package BODY LATER.

[illegible] USED BY Zone_Task_Type BODY

```
      with [illegible]

      package [illegible] Interface Package is

         subtype [illegible] Number_Subtype is Global_Data_Package.Zone_Number_Subtype;
         subtype [illegible] Degrees_Type   is Global_Data_Package.Degrees_Type;

         function Thermometer_Reading
            (Zone: Zone_Number_Subtype) return Degrees_Type;

      end Thermometer_Interface_Package;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

      with Global_Data_Package;

      package Vent_Interface_Package is

         subtype Zone_Number_Subtype is Global_Data_Package.Zone_Number_Subtype;

         procedure Open_Vent (Zone: in Zone_Number_Subtype);
         procedure Close_Vent (Zone: in Zone_Number_Subtype);

      end Vent_Interface_Package;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

      package Heater_Package is

         procedure Get_Open_Vent_Clearance;
         procedure Report_Vent_Closed;

      end Heater_Package;
```

INSTRUCTOR NOTES

THIS IS THE FIRST OF TWO SLIDES SHOWING THE Zone_Task_Type TASK BODY.

THE CONSTANT Margin IS THE NUMBER OF DEGREES ABOVE THE DESIRED TEMPERATURE AT WHICH THE VENT SHOULD BE CLOSED. THE FLAG Vent_Closed KEEPS TRACK OF THE CURRENT VENT STATUS. THE RENAMING DECLARATIONS ALLOW CERTAIN VERSIONS OF "+" AND "-" PROVIDED BY THE PREDEFINED PACKAGE Calendar TO BE USED AS OPERATORS WITHOUT A use CLAUSE FOR THE PACKAGE.

THE TASK BEGINS BY ACCEPTING A CALL ON Assign_Zone TO INITIALIZE My_Zone TO THE APPROPRIATE ZONE NUMBER. THEN Next_Sample_Time AND Current_Temperature ARE INITIALIZED. FINALLY, THE TASK ENTERS THE INFINITE LOOP SHOWN ON THE NEXT SLIDE.

SUBUNIT FOR Zone_Task_Type (SLIDE 1 OF 2)

```
with Heater_Package;
with Calendar, Thermometer_Interface_Package, Vent_Interface_Package;

separate (Zone_Package)

task body Zone_Task_Type is

   Time_Between_Samples     : constant Duration := 15.0;
   Margin                   : constant := 2.0;
   My_Zone                  : Zone_Number_Subtype;
   Current_Temperature,
      Desired_Temperature   : Degrees_Type;
   Vent_Closed              : Boolean := True;
   Next_Sample_Time         : Calendar.Time;

   function "+" (Left: Calendar.Time; Right: Duration) return Calendar.Time
      renames Calendar."+";

   function "-" (Left, Right: Calendar.Time) return Duration
      renames Calendar."-";

begin

   accept Assign_Zone (Zone: in Zone_Number_Subtype) do
      My_Zone := Zone;
   end Assign_Zone;

   Next_Sample_Time := Calendar.Clock + Time_Between_Samples;
   Current_Temperature :=
      Thermometer_Interface_Package.Thermometer_Reading (My_Zone);

   loop
          (LOOP SHOWN ON NEXT SLIDE)
   end loop;

end Zone_Task_Type;
```

INSTRUCTOR NOTES

THE LOOP FIRST EXECUTES A SELECTIVE WAIT THAT WAITS UNTIL A NEW DESIRED TEMPERATURE IS ASSIGNED OR IT IS TIME TO TAKE A NEW TEMPERATURE SAMPLE. THE DELAY ALTERNATIVE IS FORMULATED IN TERMS OF Next_Sample_Time TO AVOID CUMULATIVE DRIFT.

AS SOON AS EITHER A NEW DESIRED TEMPERATURE OR A NEW ACTUAL TEMPERATURE IS AVAILABLE -- AND ONLY THEN -- THESE TWO QUANTITIES ARE COMPARED. THE COMPARISON IS DIFFERENT WHEN THE VENT IS OPEN AND WHEN IT IS CLOSED. THE VENT IS OPENED OR CLOSED AS NEEDED.

(IN THE delay ALTERNATIVE, THE VERSION OF "-" CALLED IN THE delay STATEMENT AND THE VERSION OF "+" CALLED IN THE LAST ASSIGNMENT STATEMENT ARE THE VERSIONS PROVIDED BY Calendar and RENAMED ON THE PREVIOUS SLIDE.)

SUBUNIT FOR Zone_Task_Type (SLIDE 2 OF 2)

```
loop

   select
      accept Assign_Desired_Temperature (Temperature: in Temperature_Type) do
         Desired_Temperature := Temperature;
      end Assign_Desired_Temperature;
   or
      delay Next_Sample_Time - Calendar.Clock;
      Current_Temperature :=
         Thermometer_Interface_Package.Thermometer_Reading (My_Zone);
      Next_Sample_Time := Next_Sample_Time + Time_Between_Samples;
   end select;

   if Vent_Closed then
      if Current_Temperature < Desired_Temperature then
         Heater_Package.Get_Open_Vent_Clearance;
         Vent_Interface_Package.Open_Vent (My_Zone);
         Vent_Closed := False;
      end if;
   else  -- Vent is open.
      if Current_Temperature >= Goal_Temperature + Margin then
         Vent_Interface_Package.Close_Vent (My_Zone);
         Heater_Package.Report_Vent_Closed;
         Vent_Closed := True;
      end if;
   end if;

end loop;
```

INSTRUCTOR NOTES

Heater_Task IS DECLARED INSIDE THE Heater_Package BODY. Heater_Task HAS THREE ENTRIES. A CALL ON Request_Heat STARTS THE HEATER IF IT WAS OFF. A CALL ON Wait_For_Warm_Heater IS ACCEPTED AS SOON AS THE HEATER HAS BEEN ON FOR TWO MINUTES (PERHAPS IMMEDIATELY). A CALL ON Relinquish_Heat INFORMS Heater_Task THAT A VENT HAS BEEN CLOSED, SO THAT THE HEATER SETTING CAN BE ADJUSTED ACCORDINGLY.

THE PROCEDURES PROVIDED BY Heater_Package SIMPLY MAKE APPROPRIATE SEQUENCES OF ENTRY CALLS ON Heater_Task. WHEN Get_Open_Vent_Clearance IS CALLED BEFORE THE HEATER IS WARMED UP, EXECUTION OF THE PROCEDURE BODY WAITS AT THE SECOND ENTRY CALL.

THE PACKAGE DECLARED AT THE BOTTOM OF THE SLIDE IS USED BY Heater_Task TO CONTROL THE SETTING OF THE HEATER. WE SHALL NOT LOOK AT THE PACKAGE BODY.

THE Heater_Task SUBUNIT IS SHOWN ON THE NEXT TWO SLIDES.

BODY OF Heater_Package AND DECLARATION OF Heater_Interface_Package

```
package body Heater_Package is

   task Heater_Task is
      entry Request_Heat;
      entry Wait_For_Warm_Heater;
      entry Relinquish_Heat;
   end Heater_Task;

   task body Heater_Task is separate;

   procedure Get_Open_Vent_Clearance is
   begin
      Heater_Task.Request_Heat;
      Heater_Task.Wait_For_Warm_Heater;
   end Get_Open_Vent_Clearance;

   procedure Report_Vent_Closed is
   begin
      Heater_Task.Relinquish_Heat;
   end Report_Vent_Closed;

end Heater_Package;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

package Heater_Interface_Package is

   type Heater_Setting_Type is (Off, Low, Medium, High);
   procedure Set_Heater (Setting: in Heater_Setting_Type);

end Heater_Interface_Package;
```

INSTRUCTOR NOTES

THE CONSTANT DECLARATIONS ESSENTIALLY "RENAME" THE ENUMERATION LITERALS PROVIDED BY Heater_Interface_Package SO THAT EXPANDED NAMES ARE NOT NEEDED.

THE TABLE Required_Setting GIVES THE HEATER SETTING APPROPRIATE FOR EACH POSSIBLE NUMBER OF OPEN VENTS.

HEATER_TASK BODY (SLIDE 1 OF 2)

```
with Heater_Interface_Package, Global_Data_Package;

separate (Heater_Package)

task body Heater_Task is

   subtype Vent_Count_Subtype is
      Integer range 0 .. Global_Data_Package.Zone_Number_Subtype'Last;

   subtype Heater_Setting_Type is
      Heater_Interface_Package.Heater_Setting_Type;

   Off    : constant Heater_Setting_Type := Heater_Interface_Package.Off;
   Low    : constant Heater_Setting_Type := Heater_Interface_Package.Low;
   Medium : constant Heater_Setting_Type := Heater_Interface_Package.Medium;
   High   : constant Heater_Setting_Type := Heater_Interface_Package.High;

   Open_Vent_Count: Vent_Count_Subtype := 0;

   Required_Setting:
      constant array (Vent_Count_Subtype) of Heater_Setting_Type :=
         (0 => Off, 1 => Low, 2 .. 3 => Medium, 4 .. 5 => High);
   Heater_Warmup_Duration: constant Duration := 120.0;

begin

   loop

        (LOOP SHOWN ON NEXT SLIDE)

   end loop;

end Heater_Task;
```

THE CURRENT POSITION WITHIN THE OUTER LOOP REFLECTS THE STATE OF THE HEATER. THIS LOOP IS EXECUTED ONCE EACH TIME THE HEATER IS TURNED ON AND THEN TURNED OFF AGAIN.

AT THE TOP OF THE LOOP THE HEATER IS off. Heater_Task WAITS FOR A CALL ON Request_Heat, TURNS ON THE HEATER, WAITS TWO MINUTES (Heater_Warmup_Duration), AND ACCEPTS A CALL ON Wait_For_Warm_Heater (SO THAT THE CALLING TASK CAN MOVE ON AND OPEN ITS VENT). WE THEN ENTER THE while LOOP EXECUTED AS LONG AS THE HEATER IS ALREADY WARMED UP. THE BODY OF THIS LOOP IS REPEATED ONLY WHEN SOME VENT IS OPENED OR CLOSED. WHEN A CALL ON Request_Heat IS RECEIVED, THE ACCOMPANYING CALL ON Wait_For_Warm_Heater IS ACCEPTED IMMEDIATELY AFTERWARD (SINCE THE HEATER IS ALREADY WARM) AND Open_Vent_Count IS INCREMENTED. WHEN A CALL ON Relinquish_Heat IS RECEIVED, Open_Vent_Count IS DECREMENTED. EACH TIME Open_Vent_Count CHANGES -- AND ONLY THEN -- WE SET THE HEATER TO THE APPROPRIATE SETTING FOR THE NEW NUMBER OF OPEN VENTS. WHEN Open_Vent_Count HAS BEEN DECREASED TO ZERO, THIS SETS THE HEATER TO off (BECAUSE Required_Setting (0) = off) AND THE while LOOP IS EXITED.

AFTER ACCEPTING A CALL ON Request_Heat, Heater_Task ALWAYS ACCEPTS A CALL ON Wait_For_Warm_Heater BEFORE ACCEPTING ANOTHER CALL ON Request_Heat. THEREFORE AT MOST ONE CALLING TASK AT A TIME CAN BE BETWEEN CALLS ON THESE TWO ENTRIES. (1) IF ONE ZONE'S TASK CALLS Request_Heat WHILE THE HEATER IS OFF AND THEN ANOTHER ZONE'S TASK CALLS Request_Heat BEFORE THE HEATER IS WARMED UP, THE SECOND TASK'S ENTRY CALL REMAINS QUEUED UNTIL THE HEATER IS WARMED UP AND THE while LOOP IS ENTERED (2) WHENEVER Heater_Task ACCEPTS A CALL ON Request_Heat FOLLOWED BY A CALL ON Wait_For_Warm_Heater, BOTH THOSE CALLS MUST HAVE BEEN MADE BY THE SAME TASK.

HEATER_TASK BODY (SLIDE 2 OF 2)

```
loop

   -- Heater is off.

   accept Request_Heat;
   Heater_Interface_Package.Set_Heater (Low) ;

   -- Heater is on but cold.

   delay Heater_Warmup_Duration;
   accept Wait_For_Warm_Heater;
   Open_Vent_Count := 1;

   -- Heater is on and warm.

   while Open_Vent_Count > 0 loop

      select
         accept Request_Heat;
         accept Wait_For_Warm_Heater;
         Open_Vent_Count := Open_Vent_Count + 1;
      or
         accept Relinquish_Heat;
         Open_Vent_Count := Open_Vent_Count - 1;
      end select;

      -- Open_Vent_Count has just changed.

      Heater_Interface_Package.Set_Heater (Required_Setting (Open_Vent_Count));

   end loop;

   -- Heater is off again.

end loop;
```

INSTRUCTOR NOTES

"THIS CONCLUDES THE COURSE"

REAL-TIME PROGRAMMING IN Ada

- Ada IS A VIABLE LANGUAGE FOR REAL-TIME PROGRAMMING
  - STANDARD PROBLEMS SUCH AS CYCLIC EXECUTIVES CAN BE SOLVED IN Ada.
  - THE RUNTIME SYSTEM CAN BE ADAPTED TO SUPPORT A PARTICULAR ENVIRONMENT.

- IF A REAL-TIME PROBLEM CAN BE SOLVED EFFECTIVELY, IT CAN BE SOLVED EFFECTIVELY IN Ada.

END

FILMED

5-86

DTIC