FINAL REPORT
GIT/EES Project C-250-200

ADA EDUCATION
FOR
TECHNICAL MANAGERS

By
John F. Passafiume

Prepared for
Defense Advanced Research Projects Agency
Contract DASG60-89-C-0041
February 1981

Georgia Institute of Technology
Engineering Experiment Station
**REPORT DOCUMENTATION PAGE**

1. **REPORT NUMBER**
   - ADA097524

2. **GOVT ACCESSION NO.**
   - 9

3. **RECIPIENT'S CATALOG NUMBER**
   - 9

4. **TITLE (Name of Report)**
   - Ada Education for Technical Managers

5. **AUTHOR(s)**
   - John F. Passafiume

6. **PERFORMING ORGANIZATION NAME AND ADDRESS**
   - Computer Science & Technology Laboratory
   - Engineering Experiment Station
   - Georgia Institute of Technology, Atlanta, GA

7. **CONTRACT OR GRANT NUMBER(S)**
   - DASC66-80-C-0041
   - ARPA Order 3922

8. **MONITORING AGENCY NAME AND ADDRESS (IF DIFFERENT FROM CONTROLLING OFFICE)**
   - Ballistic Missile Defense Advanced Technology Center
   - P. O. Box 1500
   - Huntsville, AL 35807

9. **DISTRIBUTION STATEMENT (of this Report)**
   - Approved for public release, distribution unlimited.

10. **DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)**

11. **ABSTRACT**
   - This report summarizes the efforts of the Georgia Institute of Technology to develop a model course entitled "Ada Education for Technical Managers." The course was developed by the joint efforts of the Engineering Experiment Station and the Department of Continuing Education. The overall goal was to develop a set of course materials that could be provided to DoD or other interested participants at the cost of reproduction thru proliferating knowledge of the Ada language throughout the community. Two sub-goals of the program were to present the model course on two occasions to DoD personnel and to develop a

**KEY WORDS**

- Computer Programming
- Computer Software
- Programming Languages
- Software Management

**DECLASSIFICATION DOWNGRADING SCHEDULE**

**DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE**

**UNCLASSIFIED**

---

**Security Classification of This Page (When Data Entered)**

153850
videotape version of the course that would also be made available. At the request of the government one of the two course presentations was relocated from Atlanta, Georgia to Fort Belvoir, Virginia. The resulting contract modification deleted the requirement for developing the videotape version of the course as there were insufficient funds available to procure this item.
Ada Education for Technical Managers

FINAL TECHNICAL REPORT
EES/GIT PROJECT C6250
Georgia Tech Research Institute

January 15, 1981

by

John F. Passafiume, (404) 894-3417
Computer Science and Technology Laboratory
Engineering Experiment Station
Georgia Institute of Technology

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.
ABSTRACT

The goal of this project was to develop a model course in the Ada language to train technical managers in its use with embedded command and control systems. The course was developed under the guidance of the Higher Order Language Working Group's sub-committee on training and was presented to DoD technical managers at two separate sessions. It was originally intended that a videotape version of the course would be developed and made available throughout the DoD as well as industry. This effort had to be dropped due to a reduction of the available funds. Course material in the form of viewgraph transparency masters, course outline, and course notes have been provided to DARPA and are currently under review.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. TASK OBJECTIVES</td>
<td>3</td>
</tr>
<tr>
<td>III. GENERAL METHODOLOGY</td>
<td>5</td>
</tr>
<tr>
<td>IV. RESULTS AND CONCLUSIONS</td>
<td>10</td>
</tr>
<tr>
<td>V. RECOMMENDATIONS</td>
<td>13</td>
</tr>
<tr>
<td>VI. REFERENCES</td>
<td>14</td>
</tr>
</tbody>
</table>

APPENDIX - COURSE MATERIAL ADA EDUCATION FOR TECHNICAL MANAGERS
I. INTRODUCTION

To cope with the increasingly costly and difficult problem of defense system software management, the Department of Defense established the High Order Language Working Group (HOLWG) in 1975. The mission of the HOLWG was to formulate DoD requirements for high order languages, to evaluate existing languages against those requirements and to implement the minimal set of languages for DoD use. As an administrative initiative, DoD Directive 5000.29 mandated the use of HOLs in new embedded computer systems and DoD Directive 5000.31 gave an interim list of approved HOLs. The HOLWG developed a coordinated set of requirements for a common DoD HOL. The group determined that none of the existing languages fully satisfied these requirements and that a single language meeting the requirements was both feasible and desirable. The Ada language was the result of an extensive design, development and test and evaluation effort. Steps in the ongoing phase of the program include production of compilers and other tools for software development and maintenance, control of the language, and validation of compilers. It is intended that government-funded compilers and software tools as well as the compiler validation facility will be widely and inexpensively available and well maintained.

This course, Ada Education for Technical Managers, was designed to provide military contractors and end-users with the necessary background to understand the value and impact of the Ada language concepts and features. An integrated approach to Ada instruction is used in which both management and technical rationale and data are provided. The course includes motivational
and management level information required by technical managers who have the responsibility to make programming language decisions, to justify those decisions, and to assure acceptance and smooth introduction of a new programming language. In addition, sufficient technical specifics of the language such as its design philosophy, constructs and syntax are given to enable the technical manager to see the benefits of using Ada in software systems and using its sophisticated features as they were intended.

This report summarizes the efforts of the Georgia Institute of Technology to develop the model course. The course was developed by the joint efforts of the Engineering Experiment Station and the Department of Continuing Education. The overall goal was to develop a set of course materials that could be provided to DoD or other interested participants at the cost of reproduction thus proliferating knowledge of the Ada language throughout the community. Two sub-goals of the program were to present the model course on two occasions to DoD personnel and to develop a videotape version of the course that would also be made available. At the request of the government one of the two course presentations was relocated from Atlanta, Georgia to Fort Belvoir, Virginia. The resulting contract modification deleted the requirement for developing the videotape version of the course as there were insufficient funds available to procure this item.

Georgia Tech has completed the effort on this project and provided copies of all course materials to the sponsoring agency.
II. TASK OBJECTIVES

The overall project objective was to develop teaching materials to be used in a one week Ada education course for technical managers. This included a course outline, lecture notes, viewgraphs, and videotapes. The course was tailored for persons having software management and decision making responsibilities. The course described the background motivation and merits of Ada and provided sufficient exposure to the language such that course participants could perform nontrivial tasks using the Ada language. In carrying out the proposed effort, Georgia Tech performed the following tasks.

Task I - Review of Current Ada Documents

A review of reference manuals and teaching materials currently available for the Ada programming language was conducted. This task required minimal effort and time, but served to acquaint project personnel with modifications to "older" documents and the status and content of materials already under development.

Task II - Design of Ada Course Outline

GIT/EES and ICS personnel designed and specified the structure and content of the proposed model Ada language course. The design was presented to the HOLWG Advisory Committee on Ada Education and Training for comment and approval before detailed course development was initiated. The design consisted of an annotated course outline and discussion of the approach, philosophy and rationale. Drafts of the course outline were distributed to other cognizant specialists for their suggestions and comments.

Task III - Course Development
GIT/EES and ICS personnel developed the course materials required to teach Ada. Considerable attention was paid to continuity and clarity of examples and explanation and demonstration of abstract concepts and special language features. The order in which subcomponents of this task took place followed that of the outline produced in Task II. This task consumed the majority of the project time and effort.

Task IV - Presentation of Course to Government Personnel

As part of developing and evaluating the model Ada language course, EES presented the course twice to government personnel. These courses were offered on the Georgia Tech campus and at Fort Belvoir, Virginia. During the five days of the courses, instruction and workshops were conducted eight hours per day.

Task V - Presentation and Reports

Additional oral presentations (IPRs) were given during the term of the contract. A final report and briefing along with a copy of all teaching aids developed as part of this contract are being provided to DARPA.
III. GENERAL METHODOLOGY

The development of the Ada Course was based upon an integrated approach to Ada Instruction. It was determined that the form and content of the Ada course must be consistent with the goals for which Ada was developed and the methods used in this development. (See reference 5). It was understood that the future success of the Ada programming language in helping to resolve the DoD software problem would be frustrated if Ada itself were misused. Therefore, it was considered critical to provide background not only on the mechanics of using Ada features but also on the rationale for including specific features in Ada in the chosen form.

The course was designed to include the motivational and management level information required by technical managers who have the responsibility to make programming language decisions, to justify those decisions, to assure acceptance and smooth implementation of a new programming language and to meet project objectives within time and cost constraints. In addition, sufficient technical specifics of the language such as its design philosophy, constructs and syntax would be given to enable the technical manager to write non-trivial programs in Ada and equip him to direct large scale software development in Ada using its sophisticated features as they were intended.

In this way it was felt that the course would guide the participants from the more traditional style of programming and software management to the modern philosophies that are encouraged and supported by Ada. For example, the economic and reliability incentives of top-down and structured programming, strong data typing and encapsulation would be emphasized.

Many features of Ada are new to most programmers or require usage that is
different from other languages. Some of the features may not be clear from merely reading the Ada reference manual. Ada, because of its innovative approach, demands new ways of thinking and provides new capabilities for management. Many of the language innovations deserve careful presentation and appropriate emphasis. Insufficient explanation and motivation of certain features would likely lead to their misuse or disuse by both programmers and managers. Some unique Ada features and associated issues are listed below:

- strong typing
  - benefits gained through static checking
    - enhanced reliability
  - reduced cost of debugging
    - improved readability
- subtypes
  - concept of dynamic constraints
- derived types
  - added security over subtypes
- enumeration types
  - improvements to readability
- array types
  - slices
  - specification of indices with type marks
    (dynamic arrays)
- string types
  - examples of flexible string usage supported by Ada
- record types
  - protection for variants and their discriminants
    provided to prevent aliasing (enhance reliability)
- access types
  - explanation of static versus dynamic entities
    and declaration as opposed to allocation
  - lifetime of dynamic objects
  - efficiency considerations
    - using access types instead of index computations
    - changing access-variable values versus moving data
  - dangers inherent with access types
    - problems which can occur when more than one access variable refers to the same object
    - use of uninitialized access variables
- type conversion
  - why no implicitity coercion
  - qualified expressions
  - distinctions between explicit coercion and resolution of ambiguities
- aggregates
  - concept of "value"
  - positional end named notation in component association
  - distinct usage of discriminant constraints
- structured statements
  - disciplined and effective use
  - choosing the appropriate statement for a given situation
- transfer of control
  - responsible use of "exit," "goto," and "return" statements
- exceptions
  - definition
  - proper use
- implications for verifiability
- dangers - e.g. unwarranted assumptions
- proper use
  - value in verifying program correctness
  - use in validating
- formal parameter modes
  - security of static checking
  - prevention of subtle program dependencies on the particular method of parameter passing used
- overloading
  - clarification
- visibility rules
  - visibility restrictions
  - interaction with separate compilation feature
- separate compilation
  - benefits
    - individually compile and test different units of a program or software system
    - flexibility in the order of implementing units
    - minimization of cost of recompilation after changes
- generics
  - providing proven, parameterizable components for software construction
- data abstraction
  - in terms of packages and generics
- modules
  - physical and logical interfaces
  - visible and private parts of specifications
  - separation of the logical interface from the implementation
  - support of Top-Down design

It was considered to be especially important that managers know how proper use of packages can make the lower levels of developing software visible to them and allow them to control the interaction of lower program units by controlling their interfaces. Also, the ability provided by the package feature to impose intelligible organization on both software systems and software development operations must be made clear.

One of Ada's strong points is its facility for multitasking. Traditionally, multitasking has been implemented with relatively undisciplined, ad hoc methods. Processes which are inherently parallel have been forced into sequential formats due to the constraints and limitations of the programming language used. Ada, however, provides a convenient mechanism to express application situations and problem solutions in a form more closely representing their "real world" construct. For many, a fundamental introduction to
the concept of multitasking may be necessary. In addition an appreciation for the security, simplicity and flexibility of task interaction provided by the rendezvous feature of Ada should be provided.

In summary it was apparent that the traditional didactic method needed to be supplemented with new teaching aids more appropriate to Ada.

Model Course

Ada incorporates enough new programming language constructs and design concepts such that techniques employed in teaching traditional programming languages would be grossly inadequate for a satisfactory presentation of the language. As the examples of the previous section demonstrate, Ada contains a rich repertoire of new language features, many of which would be unfamiliar even to highly experienced application programmers. Therefore, it was necessary that innovative methods be developed if the material is to be presented 1) in a well organized, clear fashion and 2) in a sufficiently short period such that programming managers can afford to set aside the time to attend a course. It is believed that the demand for Ada training will be very significant in the near future and that numerous organizations, institutions and individuals will want to serve that need. All of these will be faced with the requirement to develop teaching techniques suitable for the unique features of Ada as well as to tailor the instruction to their specific intended audience.

The quality of these courses is important to the success of the Ada language in meeting its stated objectives; however, most vehicles for course quality control are not very feasible. For example DoD could control the quality of Ada training and education by 1.) undertaking the instruction responsibility or 2.) certifying courses developed and taught by others. Neither of these options would be particularly attractive to an organization...
that is neither staffed nor chartered to perform these functions. Another possible vehicle for quality assurance is to provide a DoD approved model course to anyone wishing to develop a course in Ada. The model course was intended to be a good exemplar for those wanting to develop their own innovative teaching methods and a needed supplement for those who lack either the time or desire to undertake such an endeavor. In either case an acceptable foundation on which to build specialized courses would be available. EES proposed to develop such a course in close interaction with the HOLWG Advisory Committee on Ada Education and Training. The product of this development effort was to be a set of approved teaching materials and aids to be used in a five day training course; a course outline, lecture notes and viewgraphs, class hand-outs, sample problems and 15 hours of video taped lectures. All of these materials would be delivered to DARPA and thereafter be in the public domain.

In addition to the model course a set of realistic examples of Ada programs would provide a valuable teaching aid. Many such examples were obtained from Ada Test and Evaluation (T&E) participants and from others developing Ada courses. Additional examples were developed as a result of interactions with the Ada Education and Training Advisory Committee.
IV. RESULTS AND CONCLUSIONS

The initial guidance to Georgia Tech for the development of the course was provided on February 6, 1980 during a meeting of the Ada Education and Training Advisory Committee (see Attachment I). The committee performed a detailed review of the Georgia Tech model course and agreed that the course was well into the design phase. (These early efforts were financed by Georgia Tech as it was felt that such an important endeavor was worthy of our support.) Several recommendations were made, and it was agreed that Georgia Tech would provide a new syllabus at the next level of detail with supporting words describing the proposed examples and approach. Although it was agreed that top-down decomposition would be an excellent way to introduce concepts, it was generally agreed that the participants first needed an understanding of the basic facilities and control structures. It also appeared desirable that a set of machine readable, documented examples be collected. Finally, it was agreed that the success of courses would be enhanced by the availability of a translator, even if inefficient, so that students can get a few programs running.

An Ada Model Course review was held at Georgia Tech on April 28, 1980. During this meeting, representatives from the Ada Education and Training Committee were presented with a revised course outline and also reviewed several proposed examples intended for use during the course. As a result of this meeting and ensuing discussion, further changes were made to the course material.

Another training meeting was held in Washington, D. C. on May 13, 1980. The two primary instructors and course material developers for Georgia Tech
attended this meeting. The material was generally well received, and it was agreed that the content and direction of the course was appropriate.

Shortly after this meeting, Georgia Tech was asked by the sponsor to consider moving the first of the two courses for DoD personnel from the planned location at Georgia Tech to Fort Belvoir, Virginia. The stated reason for this move was the shortage of travel funds in DoD. The sponsor was advised that the funds remaining in the project could not allow for that move and also cover the cost of developing the planned videotape version of the course. The sponsor decided to defer development of the videotape version of the course. A contract modification was subsequently issued cancelling the videotape effort and directing that the first of the two DoD presentations be moved to Fort Belvoir, Virginia.

The first of the two contract courses was presented at Fort Belvoir, Virginia on 23–27 June 1980. One of the secondary purposes of the presentation was to provide a live audience for field testing the material. Most of this aim was accomplished during the weeks' presentation. Many of the comments were constructive and enabled Georgia Tech to provide for changes to the material. Georgia Tech feels that the course could have been improved if a software engineering approach had been used in its development.

Version control proved to be a major problem with the materials, especially since the language was not stable during the development phase and Georgia Tech was constantly being required to react to changes. This had considerable impact on costs, and funds for the remaining development ran out before final preparation of the course material had been completed.

The second of the two contract courses was presented at Georgia Tech from July 7–11 1980. The course was attended by 13 DoD personnel including the
members of the Ada Education and Training Committee. In general, the course went well and participants were receptive to the material and methodology. A comment session was conducted on July 11 and the comments were, for the most part, quite positive. The attendees were generally satisfied with the course handouts and the visual aids. Most students felt, that as an overview for managers, the course contained too much detail and too much programming. Although the attendees were purported to be software managers, they professed not to be interested in the programming details. (This is not consistent with our view as to what software managers need to know to manage a large software project and is a source of some concern if this is a prevalent view throughout DoD.)

From the staff's viewpoint, Georgia Tech felt that the material was presented at the proper level for industry technical managers. The reordering of the material resulting from the comments obtained from the first presentation at Fort Belvoir appeared to be quite successful. The instructors were more comfortable with the material and felt that the presentation went more smoothly as a result of the changes. The committee representative indicated that he was pleased with the course and felt it satisfied most of his needs. He also recognized that it was a management course and felt that the level and thrust of the presentation was quite appropriate.

On July 23, 1980, DARPA was provided with a then current set of all training materials. Constant changes and delays in reception of the final reference manual had severe impact on cost and schedule. Georgia Tech received the final copy of the reference manual in August 1980 and made applicable changes to the course material. Copies of all deliverables were provided to DARPA in September-October 1980.
V. RECOMMENDATIONS

As we have not been provided with the results of the review of the course
text material, we are unable to comment on any inputs received from the reviewers.

However, based upon our experience in the development and presentation of the
course to two DoD classes and two additional sessions under the auspices of
the Department of Continuing Education, the following recommendations are
provided:

a. Our experience in the development and presentation of the
course to two DoD and two Continuing Education classes have
shown that the course approach was valid. Therefore, future
courses in the teaching of Ada to DoD personnel should use this
course as a model.

b. The availability of a translator would have greatly enhanced
the value of the course. For an executive overview or manager
course it would have been an invaluable aid to understanding.
For a programmer’s course a translator would be a necessity.
Therefore, all future courses should include the use of some
sort of translator. The NYU translator and interpreter will
shortly be available from the U. S. Army and should be consid-
ered as a vehicle to satisfy this requirement.

c. The interaction with the Ada Education and Training Committee
was very useful and should be an element in the development of
any future Ada courses.

d. DoD should continue to explore the possibility of developing a
videotaped version of the course. User agencies/activities
could then supplement such a standard package with material
germane to their own specific requirements.

e. Georgia Tech spent considerable in-house time and effort in
the investigation of the use of color graphics for course
visuals. It is felt that this methodology offers significant
promise and future courses should consider its use, providing
the costs can be kept to a reasonable level.

f. A set of realistic examples of Ada programs would provide an
invaluable teaching aid. The development of such examples
should continue to be encouraged by the Ada Joint Project
Office. These examples should be provided to interested user
agencies at their request.
VI. REFERENCES


COURSE MATERIAL

ADA EDUCATION FOR TECHNICAL MANAGERS
TABLE OF CONTENTS

Introduction to Ada

Example I - INTRODUCTORY EXAMPLE
  Program structure, lexical units, declarations, basic statements

Example II - PROCEDURES AND FUNCTIONS
  Declaration and parameter modes, blocks, visibility, type declarations, statements, type equivalence, operators and operands

Example III - RECORD HANDLING
  Packages, records and record aggregates, case statement, input-output, program structure, visibility, separate compilation

Example IV - ENUMERATION TYPES
  Enumeration types, array aggregates, named parameter association

Example V - OVERLOADING AND EXCEPTIONS
  Overloading, exceptions, packages and exceptions

Example VI - LIST PROCESSING
  Access types, data abstraction, generics, discriminants, variant records

Example VII - FUNDAMENTALS OF TASKING
  Task concepts

Example VIII - TASK INTERACTIONS
  Entries, accept statements, rendezvous, task attributes, select statements

Case Study I - PROGRAM DESIGN USING PACKAGES
  A Text Formatter

Case Study II - USING TASKS FOR SIMULATION
  Telephone Switching Simulation

Case Study III - REAL TIME CONTROL
  HELBAT BIFF

Summary
1.0 Abstract

The emerging DoD programming language, Ada, promises to aid the software developer by offering capabilities formerly considered outside the scope of an HOL. Ada is PASCAL-like in its design and includes such modern programming concepts as strong data typing, blocking and hierarchical exception handling. In addition to the capabilities found in modern HOLs, Ada includes some relatively new areas: real time processing, libraries, and assertions.

This paper compares the present military standard languages, JOVIAL J73, CMS-2 and FORTRAN, to Ada in seven areas: design criteria, general syntax, data typing, control, functions, real-time processing, and other advanced techniques. This comparison shows which areas are new to the HOL arena, and how modern programming techniques have been used to increase the applicability and reliability of traditional HOL areas.

2.0 HOL Feature Study

The HOL Feature Study compares several HOLs at a functional level and obtains their relative ranking. Sections 3.0 and 4.0 give the basis for our selection of the four HOLs analyzed in the feature study; Section 5.0 explains the methodology used in the feature study; and Section 6.0 analyzes the results of these scores. The primary conclusions of the feature study are as follows:

- All four languages encourage the use of structured programming control constructs.
- Ada and J73 provide strong typing (i.e., grouping data of like value ranges and operations under specific type names) which allows significant reliability improvements in code.
- Although Ada is top rated according to features, it will not be easy to learn and it presents some difficulties to compiler implementors.

3.0 History of High Order Languages

HOLs were originally developed in the late fifties to present mathematical algorithms to a computer. FORTRAN, one of the earliest, was designed expressly to translate numeric formulas. ALCOL was somewhat more elegant in its approach, but again its primary purpose was the expression of mathematical algorithms. The Air Force subsequently developed JOVIAL by tailoring much of ALCOL to the needs of command and control systems. Similarly, the Navy developed CMS-2 because it needed more capability than the conventional languages possessed.

These early languages grew as the computer technology grew. The old familiar languages were modified and extended to meet new situations, often situations not anticipated in the original language. COBOL was defined to address the problems of business data processing where mathematics is limited to financial areas and there is a stronger need for manipulation and display of character data. PL/I was defined in the mid 60's in an attempt to bridge the scientific and business application areas. As such it included the major features of COBOL as well as those of FORTRAN, ALCOL, and JOVIAL.

All six of these major programming languages were designed primarily as improvements in power and capability over existing languages. Benefits to programmers were derived from the continuous addition of new features.

Not until the definition of PASCAL in 1971 was a language formulated whose primary design goal was to aid the programmer in developing his software. It includes a rich set of data types and allows only a small set of control structures
supporting structured programs. The advent of PASCAL represents the point in computer science when sufficient understanding of HCL problems and beneficial HUL techniques had been accumulated to address the proper engineering of these languages. The emphasis shifted from expressing the problem for the machine to expressing the problem for the programmer. Ada builds the philosophy of PASCAL into a language powerful enough to support large software systems.

4.0 High Order Languages Selected

The evolution and major characteristics of the languages selected for the HOL Feature Study can be discussed against this general background. FORTRAN, JOVIAL, and CMS-2, three widely used languages from the DoD list of approved high order languages, were originally selected for this study. When the DoD common high order language effort remained on schedule and a single language design for Ada was chosen in May, 1979, it was decided to also evaluate Ada to keep the HOL Feature Study at the state of the art.

The next step after selecting the four languages was to choose the exact dialect for each language. Each language presented its own special problems. Standardization has been a major problem in the use of high order languages since their inception. Throughout their history proliferation of dialects and language derivations have occurred in spite of on-going standardization efforts. Recently these standardization efforts have become increasingly stronger within the Department of Defense and are just now beginning to exhibit results.

Although a standard for FORTRAN IV was established in 1966, most current FORTRAN compilers support a superset of FORTRAN IV. For example, the extensions to FORTRAN supported by these compilers incorporate more modern structuring techniques, reduce the rigidity of the fixed formats for statements and comments, and provide character manipulation facilities. Although there is not a consensus in the selection or the implementation of these extensions among the various compilers, it would be unrealistic to limit the evaluation of FORTRAN to the 10 year old standard FORTRAN IV subset. In 1977, the ANK1 standard for FORTRAN was updated to include several of the more common extensions such as improved /O, blocked IF-THEN-ELSE, character string manipulations and wider use of expressions in place of integers. However, few compilers that support FORTRAN 77 are currently available. In order to reflect these improvements, the FORTRAN feature study analysis is based on FORTRAN 77 (Ref.2) and FFP (Ref.3), the FORTRAN compiler for PDP-11's, which is representative of commonly available extended FORTRAN compilers.

The JOVIAL language has been used by the Air Force since 1959. It is a derivative of ALGOL and was specifically modified to support command and control systems. As an ALGOL derivative it contains the blocked structures necessary for structured programming and has had a more freely formatted source input than FORTRAN. Additions for command and control systems include low-level rather than high-level /O, logical decision making based on flags, and the capability to build large systems consisting of several independently compiled modules.

Like FORTRAN, JOVIAL also suffered from the proliferation of dialects and minor differences between implementations. In 1967, a version of JOVIAL J3 was established by the Air Force as its standard programming language for command and control systems. In 1972 a committee report was accepted to modernize J3. The new dialect, J73/1, was adopted as the official Air Force standard, but a JOVIAL implementation called J3B was developed based upon a preliminary report from the modernization committee. Due to schedule considerations, J3B was used on several operational flight programs (F16 and B-1) and underwent further modifications picking up strong typing rules and tighter control of inter-compiler unit interfaces. In late 1978, the Air Force undertook an effort to standardize on a single dialect of JOVIAL by incorporating the proven capabilities of J3B into the otherwise more modern J73/1. The result of this effort is known as J3 and contains improvements over both J73/1 and J3B. The MIL-STD-1558A definition of JOVIAL (J73) (Ref.4) has become the official Air Force standard and has been selected for evaluation under the HOL feature study.

The Navy has taken a much stronger approach to the control and standardization of their language, CMS-2. It is based upon the Compiler System-1 (CS-1) first used by the NAVY circa 1955. When it was decided in 1964 to upgrade CS-1, the task of coordinating the effort was given to what is now the Fleet Combat Direction System Support Activity (FCDSSA). FCDSSA has complete control over the generation and distribution of all CMS-2 compilers within the Navy. In upgrading CS-1, it was decided to include the best features of existing languages while maintaining as much compatibility as possible with existing CS-1 programs. As a result CMS-2 includes the features of structured programming and the ability to specify packed tables for interfacing with hardware defined data structures, but it also contains more primitive constructs which are often redundant. (Ref.5)

In addition to rigorously controlling the CMS-2 language, the Navy has also standardized on the processors to be used in its systems. The AN/UYK-7 is a large mainframe and the two mini-computer families used are the AN/UYK-20 and the CP-642. Thus while CMS-2Y represents a significant update to CMS-2, CMS-2M is merely the tailoring of CMS-2Y to the AN/UYK-20 processor. Although CMS-2 is fairly machine independent, CMS-2M documentation gives the impression of machine dependency because of the processor standardization and the strong hardware/software association. Because it is the most recent language definition, CMS-2M, as defined in the M-5045 CMS-2Y (20) User Manual (Ref.6), was chosen for the feature study analysis.
Unlike the other three languages, Ada has a very short history. It is the result of an intensive effort to standardize on a single language for embedded computer systems throughout DoD. The High Order Languages Working Group (HOLWG) was organized in 1975. In reviewing the existing languages, the HOLWG found that no existing language satisfactorily met their broad range of requirements. The HOLWG then began successively refining the language requirements over a four year period. This process was highly interactive, receiving inputs from numerous contractors as well as the individual military branches. Four preliminary PASCAL-like language designs were evaluated and the language design narrowed to two candidates, called RED (Ref.7) and GREEN (Ref.8). The two design teams modified these languages according to the final requirement specifications found in the STEELMAN (Ref.9) document. As a result of an intensive evaluation by both contractors and military teams, the GREEN language design for Ada was selected in May of 1979. This will undergo a test and evaluation period during which tests was run on an Ada simulator. Final revisions to the Ada language definition will be made in early 1980. The Ada language as defined by the March 15 Reference Manual for the GREEN Programming Language will be evaluated under the HOL feature study.

5.0 HOL Feature Study Methodology

The common HOL language effort has resulted in another major contribution to the HOL Feature Study. The set of features used to compare FORTRAN, JOVIAL J73, CMS-2K, and Ada is based upon the STEELMAN language requirements. STEELMAN represents the culmination of years of intensive discussion and interaction of literally thousands of high order language users and experts. We have reviewed these requirements, selecting 6 general goals and 46 specific language features required by embedded computer systems.

Project members independently weighted the 52 features from one to ten according to the feature's importance with respect to general programming requirements. After discussion, each feature was assigned a general weight by group consensus. Table I lists the 52 features, their associated paragraphs in STEELMAN and their maximum programming weights.

Having thus arrived at a maximum score for each feature, specific scoring criteria were developed to further quantify the analysis and to facilitate consistency across language evaluations. The scoring criteria were each assigned relative values so that their relative importance was maintained and their totals equaled the maximum allotted to the feature. Finally, independent evaluations were performed on Ada, J73, CMS-2 and FORTRAN.

6.0 Feature Study Results

By quantifying the scoring as much as possible and selecting specific scoring criteria, much of the HOL feature study effort was accomplished by the comparison approach. With most features, determining the number of points a particular language should receive was straightforward. Even though languages were scored by more than one reviewer, general agreement occurred on the first pass and minor differences were quickly resolved. Each feature was resolved into a number of scoring criteria which were evaluated independently. For example, the "Bit Strings" feature was broken into assignment; equivalence or non-equivalence; complement; intersection, union and symmetric difference; and set membership (substrings). These were each assigned a maximum value of 2 or 3 and the languages were each scored on that range for that criterion. These results were summed to give the final score for that feature.

The remainder of this section correlates the resulting feature study scores with the conclusions stated earlier in the introduction to Section 2.6. Tables I and II provide a summary of the raw scores and a grouping of the individual feature scores into more general categories.

The totals from Table I give an ordering of the power of the four languages studied. The ordering (from weakest to strongest: FORTRAN, CMS-2, J73, Ada) is not surprising. FORTRAN is the only one not specifically designed for military systems. Ada represents the most recent language design theory and had the STEELMAN requirements as a guideline. The higher score of J73 over CMS-2 reflects the inclusion of stronger typing, exception handling, and stricter parameter matching in the recent J73 upgrade.

Differences between the languages are explained in greater detail in Sections 6.1 thru 6.4, which cover each language individually.

Before discussing the language differences, we should point out the commonality among the languages. With the revisions made in the 1977 version of FORTRAN, all four languages now support structured programming. This is reflected by the relatively high subtotals for the CONTROL category in Table II. The point is further made that FORTRAN and CMS-2 were penalized primarily for the lack of short circuiting (not really part of structured programming) and minor shortcomings with respect to WHILE loops and LOOP EXITS. (Refer to Features 23 to 30, Table I.)

In fact, if the scores were adjusted to disregard strong typing, real time processing, exception handling, and separate translation facilities, the scores for all four languages would be relatively consistent. This is not to say that these features are not important. They represent the major improvements made by Ada and
<table>
<thead>
<tr>
<th>FEATURE</th>
<th>STEELMAN</th>
<th>ADA FOR J73 CMS MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESIGN CRITERIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Reliability</td>
<td>A,B</td>
<td>10</td>
</tr>
<tr>
<td>2. Maintainability</td>
<td>C</td>
<td>5</td>
</tr>
<tr>
<td>3. Efficiency</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>4. Simplicity</td>
<td>E</td>
<td>6</td>
</tr>
<tr>
<td>5. Machine Independence</td>
<td>G</td>
<td>10</td>
</tr>
<tr>
<td>6. Complete Definition</td>
<td>H</td>
<td>10</td>
</tr>
<tr>
<td>GENERAL SYNTAX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. General Syntax</td>
<td>A,B,D</td>
<td>7</td>
</tr>
<tr>
<td>8. Syntactic Extensions</td>
<td>C</td>
<td>5</td>
</tr>
<tr>
<td>9. Identifiers</td>
<td>E,F</td>
<td>5</td>
</tr>
<tr>
<td>10. Literals</td>
<td>G,H</td>
<td>5</td>
</tr>
<tr>
<td>11. Comments</td>
<td>I</td>
<td>9</td>
</tr>
<tr>
<td>12. Strong Typing</td>
<td>3A,B,D</td>
<td>10</td>
</tr>
<tr>
<td>13. Type Definitions</td>
<td>3C,D</td>
<td>10</td>
</tr>
<tr>
<td>14. Numeric Types</td>
<td>3A,D-H</td>
<td>10</td>
</tr>
<tr>
<td>15. Numeric Operations</td>
<td>3B,C</td>
<td>10</td>
</tr>
<tr>
<td>16. Enumeration Types</td>
<td>3D,A,B</td>
<td>10</td>
</tr>
<tr>
<td>17. Boolean Type</td>
<td>3C</td>
<td>10</td>
</tr>
<tr>
<td>18. Character Types</td>
<td>3D</td>
<td>10</td>
</tr>
<tr>
<td>19. Arrays</td>
<td>33A-E</td>
<td>10</td>
</tr>
<tr>
<td>20. Records</td>
<td>33F-H</td>
<td>10</td>
</tr>
<tr>
<td>21. Indirect Types</td>
<td>33I,J</td>
<td>10</td>
</tr>
<tr>
<td>22. Bit Strings</td>
<td>34A,B</td>
<td>10</td>
</tr>
<tr>
<td>23. Encapsulation</td>
<td>35A,B</td>
<td>10</td>
</tr>
<tr>
<td>24. Scoping</td>
<td>35C,5C,G,7C</td>
<td>10</td>
</tr>
<tr>
<td>25. Declarations</td>
<td>5A,B,D,F</td>
<td>10</td>
</tr>
<tr>
<td>26. Initial Values</td>
<td>5E</td>
<td>10</td>
</tr>
<tr>
<td>27. Expressions</td>
<td>5A-G</td>
<td>10</td>
</tr>
<tr>
<td>DATA TYPING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28. Control Structures</td>
<td>6A,B</td>
<td>10</td>
</tr>
<tr>
<td>29. Conditional Control</td>
<td>6A,C</td>
<td>10</td>
</tr>
<tr>
<td>30. Iterative Control</td>
<td>6A,E</td>
<td>10</td>
</tr>
<tr>
<td>31. Explicit Transfer</td>
<td>6A,G</td>
<td>10</td>
</tr>
<tr>
<td>32. Short Circuiting</td>
<td>6D</td>
<td>10</td>
</tr>
<tr>
<td>CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33. Procedures</td>
<td>7A,D</td>
<td>10</td>
</tr>
<tr>
<td>34. Recursion</td>
<td>7B</td>
<td>10</td>
</tr>
<tr>
<td>35. Parameter Passing</td>
<td>7P-H</td>
<td>10</td>
</tr>
<tr>
<td>36. Aliasing</td>
<td>7I</td>
<td>10</td>
</tr>
<tr>
<td>37. Low Level I/O</td>
<td>8A,E</td>
<td>10</td>
</tr>
<tr>
<td>38. Hi Level I/O</td>
<td>8F,C,D,F</td>
<td>10</td>
</tr>
<tr>
<td>FUNCTIONS I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39. Parallel Processing</td>
<td>9A,B,H,I,J</td>
<td>10</td>
</tr>
<tr>
<td>40. Mutual Exclusion</td>
<td>9C</td>
<td>10</td>
</tr>
<tr>
<td>41. Scheduling</td>
<td>9D</td>
<td>10</td>
</tr>
<tr>
<td>42. Real Time</td>
<td>9E</td>
<td>10</td>
</tr>
<tr>
<td>43. Interrupts</td>
<td>9E</td>
<td>10</td>
</tr>
<tr>
<td>44. Async. Termination</td>
<td>9C</td>
<td>10</td>
</tr>
<tr>
<td>REAL TIME PROCESSING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45. Exception Handling</td>
<td>10A-E,G</td>
<td>10</td>
</tr>
<tr>
<td>46. Assertions</td>
<td>5F</td>
<td>10</td>
</tr>
<tr>
<td>47. Data Representation</td>
<td>11A</td>
<td>10</td>
</tr>
<tr>
<td>48. Lang. Interface</td>
<td>11E</td>
<td>10</td>
</tr>
<tr>
<td>49. Optimizations</td>
<td>11C,D,F</td>
<td>10</td>
</tr>
<tr>
<td>50. Libraries</td>
<td>12A</td>
<td>10</td>
</tr>
<tr>
<td>51. Separate Trans.</td>
<td>12B</td>
<td>10</td>
</tr>
<tr>
<td>52. Generic Definitions</td>
<td>12D</td>
<td>10</td>
</tr>
<tr>
<td>OTHER TECHNIQUES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTALS</td>
<td>373 177 290 210 394</td>
<td>394</td>
</tr>
</tbody>
</table>

Table 1. Functional Comparisons
to a lesser extent J73. The point to be made is that the remaining features would represent functional capabilities sufficient for many problems. All four languages provide these functions with only minor improvements made in J73 and Ada. The additions made by these languages don't provide new capabilities, but rather, allow the programmer to state the solution in a more precise, reliable, and straightforward manner. These characteristics are precisely those that will aid maintenance efforts and reduce life cycle costs.

6.1 FORTRAN

Although FORTRAN contains the basic functional capabilities required by many problems, FORTRAN programmers would encounter difficulties in several areas: low-level I/O, partial word data, and tightly packed records.

Features 37 and 48 are those most pertinent to low-level I/O. As can be seen, FORTRAN contains no provisions for explicitly specifying low-level I/O instructions. These must be implemented as calls to assembly language routines. Since I/O operations normally entail only a single machine instruction, subroutine linkage overheads of 3 to 4 words represent significant increases. A much greater problem occurs on time-critical I/O operations (e.g., disable interrupts) which can't allow any intervening overhead instructions.

FORTRAN is unable to specify data items requiring less than a full word or byte of memory, as is indicated by features 47, 22 and 16. In order to access specific bit strings within a word, the programmer must use explicit masking and shifting operations. In addition to being error prone, this makes code less understandable because descriptive names cannot be associated with specific bits.

6.2 CMS-2

CMS-2 corrects most shortcomings found in FORTRAN. Specified tables may contain items of different types and may assign exact sizes and bit positions to individual items. Using these features, the CMS-2 programmer can access each field by an appropriate variable name. Low-level I/O in CMS-2 is accomplished by allowing insertion of assembly language directly between CMS-2 statements. Although these features are not controlled as well as the corresponding features in Ada and J73, they allow many military software systems to be well represented in CMS-2.

The major shortcomings in CMS-2 are its lack of strong typing and the presence of outdated features. This second characteristic was caused by the decision to maintain downward compatibility of compilers. It results in special cases and duplicated features throughout CMS-2. The 150-plus keywords found in CMS-2 are indicative of its complexity for both implementation and maintenance programmers. Secondly, CMS-2 is comparatively weak in data typing. Scoping is less powerful; some data types are either missing, as in the case of enumeration types, or are restricted, as in the case of bit strings; and the user is not allowed to group data by defining his own types. These features are desirable to facilitate code reliability.

6.3 JOVIAL (J73)

Table 1 shows that J73 consistently outscores CMS-2. The number and types of constructs found in J73 have been greatly condensed without losing any of the functional capability found in CMS-2. Beyond CMS-2, J73 has included the basis for strong typing, fundamental exception handling, tighter control of functions and procedures, and slight improvements in control structures. The strong typing and exception handling capabilities of J73 were adopted from early work on Ada and as such are not nearly as well developed as those in Ada. The four areas mentioned here account for most of the 50 point difference between J73 and CMS-2. The overall effect of these features is an increase in reliability and maintainability as indicated in features 1 and 2 of the General Design Criteria section. (Table 1)

J73's major improvements in control structures are loop EXITS and short circuiting of conditional expressions. Loop EXITS provide a controlled alternative to explicit GO TO's or match flags for exiting iterative loops upon the occurrence of desired conditions. Short circuiting allows the use of logical properties to optimize complex decisions. For example, the decision

IF A=0 or B=0 or (C=0 and D=x1)

is known to be true as soon as A is found to equal zero, and the remaining conditions need not be checked.

J73 introduces several improvements to functions and procedures. Strong parameter type checking is supported across separate compilation, as well as within compilation units. Machine specific functions and procedures allow a well controlled means of introducing low-level I/O. J73 compilers will recognize a special set of what look like procedure or function calls as requesting inline generation of machine specific instructions. Recursive procedures are also supported. These improvements to functions and procedures allow compile-time error detection in this area and result in more reliable code.

Another J73 improvement related to procedures and functions is the abort capability covered in feature 45. An alternate return may be specified on procedure calls. Execution of the ABORT statement within called procedures will subsequently return control to the most recently specified alternate return. This provides an efficient means of handling error conditions without destroying the single-entry-single-exit benefits of structured programming.

The most important reliability improvements in J73 are obtained from its strong typing features. This is reflected by J73's 26 point
I. Introduction

In this section Ada's scores in the Design Criteria and Data Typing areas of Table I will be compared with those of the second place Digital Equipment Corporation (J73) in order to determine whether Ada is a suitable language for use in a real-time executive. The impact of strong typing in Ada is so great that the high scores it receives are also a measure of the reliability of the code produced by Ada.

6.4 Ada

Ada takes the benefits found in J73's strong typing one step further. Strong data typing is a fundamental characteristic of Ada. In addition to user definable types, Ada provides sub-types to specify absolute value ranges which are automatically checked across all assignments. Moreover, most features in Ada contain nuances which reflect the assumption of very strong data typing. Overloading of procedures, encapsulation, and generic program units are examples of new concepts in Ada highly associated with strong typing. The impact of strong typing in Ada is so dominant as to force a new style of programming. This new approach greatly enhances the production of reliable code. These capabilities are indicated by Ada's high scores in the Design Criteria and Data Typing areas of Table I.

While providing this radical departure from the other three languages, Ada consistently builds upon their proven capabilities. Comparing the Ada scores in Table I with those of the second place language, J73, we find 35 features in which Ada receives a higher score and only 5 in which it scores lower. In these five features the Ada score is lower by only a single point in each case.

The second area of significant improvement in Ada is the inclusion of real time processing constructs. In this section of Table II, Ada receives almost a full score while the other languages receive almost no points at all. The Ada language contains the fundamentals of a real time executive. Presently such executives are implemented via several routines particular to each operating system. In Ada, desired executive control and synchronization of independent tasks can be obtained by proper selection of built-in language constructs. Incorporation of these features directly in the language not only reduces implementation efforts but also establishes a consistent approach across systems.

Ada's score of 373 out of a possible 394 points clearly marks it as the most desirable language choice. There are a few reservations, however, concerning Ada due to its early stage of development. Ada has just been defined as of March, 1979, and is still undergoing refinement. No Ada compiler has yet been implemented. As we have discussed above, Ada imposes a new style of HOL programming. It includes many new features unfamiliar to a large segment of programmers. While providing many benefits, these features will require a learning process. They also present new implementation problems to compiler designers. Certainly, additional complexity should be avoided in any changes made during the Ada test and evaluation process and the importance of initial compiler implementation efforts should not be underestimated.

<table>
<thead>
<tr>
<th>ADA FOR</th>
<th>J73</th>
<th>CMS MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Criteria</td>
<td>49</td>
<td>35</td>
</tr>
<tr>
<td>General Syntax</td>
<td>35</td>
<td>28</td>
</tr>
<tr>
<td>Data Typing</td>
<td>111</td>
<td>47</td>
</tr>
<tr>
<td>Control</td>
<td>50</td>
<td>33</td>
</tr>
<tr>
<td>Functions &amp; I/O</td>
<td>43</td>
<td>19</td>
</tr>
<tr>
<td>Real Time Processing</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>Other Techniques</td>
<td>57</td>
<td>15</td>
</tr>
<tr>
<td>Totals</td>
<td>373</td>
<td>177</td>
</tr>
</tbody>
</table>

Table II. Summary of Results

References

An Introduction to Ada
Course Outline

FIRST DAY
Overview of Ada
History of Ada, comparison to present military standard languages, introduction to Ada Features

Example I - Introductory Example
Program structure, lexical units, declarations, basic statements

Example II - Procedures and Functions
Declaration and parameter modes, blocks, visibility, type declarations, statements, type equivalence, operators and operands

SECOND DAY
Example III - Record Handling
Records and record aggregates, packages, case statement, input-output, program structure, visibility, separate compilation

Example IV - Enumeration Types
Enumeration Types, Array aggregates, named parameter association

Case Study I - Program Design Using Packages

THIRD DAY
Example V - Overloading and Exceptions
Overloading, exceptions, exceptions in packages

Example VI - List Processing
Access types, data abstraction, generics, discriminants, variant records

Case Study II - Real Time Control - Overview

FOURTH DAY
Example VII - Fundamentals of Tasking
Task concepts

Example VIII - Task Interactions
Entries, accept statements, rendezvous, task attributes, select statements

Case Study II - Real Time Control - Implementation

Summary
ADA INTRODUCTION

SYNTAX
- designed for readability

DECLARATIONS and TYPES
- factorization of properties, maintainability
- abstraction, hiding of implementation details
- reliability, due to checking
- floating point and fixed point, portability
- access types, utility and security

STATEMENTS
- assignment, iteration, selection, transfer
- uniformity of syntax (comb structure)
- generally as simple as possible
  (e.g., iteration control)

SUBPROGRAMS
- procedures and functions
- logically described parameter modes
  (as opposed to definition by
  implementation description)
- overloading

PACKAGES
- modularity and abstraction
- structuring for complex programs
- hiding of implementation, maintainability
- major uses:
  . named collections of declarations
  . groups of related subprograms
  . encapsulated data types
LIBRARIES
- separate compilation
- generics
- program development environment

TASKING
- can be done completely with Ada features
- single concept for intertask communication and synchronization
- interface with external devices
- designed for efficient implementation

EXCEPTION HANDLING
- for reliability of real-time systems
- standard vs. user-defined exceptions
- meant mainly for handling errors (rather than as a general programming technique)

MACHINE DEPENDENCIES
- representation specifications
- interface with other languages
- low level I/O
Ada is designed for writing large programs.

Ada has features to allow suitable extensions for a particular application.

Ada is a design language.
EXAMPLE I

INTRODUCTORY EXAMPLE
OBJECTIVES

Program Structure
Lexical Units
Declarations
Basic Statements
LOGICAL STRUCTURE

with TEXT_IO;

procedure MIN_MAX_SUM is

   declarative part

begin

   sequence of statements

end MIN_MAX_SUM;
with TEXT_IO;

procedure MIN_MAX_SUM is

    ...

begin

    ...

    for ... loop

        ...

        if ... then

            ...

        elsif ... then

            ...

        end if;

        ...

    end loop;

    ...

end MIN_MAX_SUM;
A COMPLETE PROGRAM

with TEXT_IO;
procedure MIN_MAX_SUM is

-- This program reads a list of one or more integers and
-- reports the minimum, maximum, and sum of them. The
-- program expects this list to be preceded by an integer
-- value giving the number of integers in the list.

use TEXT_IO;
ITEM : INTEGER;
MAXIMUM : INTEGER;
MINIMUM : INTEGER;
SUM : INTEGER;
NUMBER_OF_ITEMS : INTEGER range 1..INTEGER'LAST;

begin
GET(NUMBER_OF_ITEMS);       -- Read the length of the list
                            -- Assume NUMBER_OF_ITEMS >= 1
GET(ITEM);
MAXIMUM := ITEM;
MINIMUM := ITEM;
SUM := ITEM;

for N in 2..NUMBER_OF_ITEMS loop  -- Loop variable is
    -- declared automatically
    -- Its scope is range of
    -- loop statement
        GET(ITEM);
    if ITEM > MAXIMUM then
        MAXIMUM := ITEM;
    elsif ITEM < MINIMUM then
        MINIMUM := ITEM;
    end if;

    SUM := SUM + ITEM;
end loop;

PUT(" MAXIMUM IS ");  PUT(MAXIMUM);  NEW_LINE;
PUT(" MINIMUM IS ");  PUT(MINIMUM);  NEW_LINE;
PUT(" SUM IS ");  PUT(SUM);  NEW_LINE;

end MIN_MAX_SUM;
LEXICAL UNITS

IDENTIFIERS
RESERVED WORDS
NUMBERS
STRINGS
DELIMITERS

- any number of spaces between lexical units
- at least one space between adjacent identifiers
- or numbers
IDENTIFIERS

MIN_MAX_SUM -- underscore is significant
MINMAXSUM -- not the same as MIN_MAX_SUM

ITEM

NUMBER_OF_ITEMS -- no distinction made
Number_Of_Items -- between upper and
-- lower case

Size_30 -- identifier may include digits

-- Composed of letters, digits, and
-- isolated underscores
--
-- First character must be a letter
--
-- Last character must be a letter
-- or a digit
--
-- All characters are significant;
-- length of identifier restricted
-- only by length of line
RESERVED WORDS

procedure is
begin
end
if then else elsif
for in loop
(not a complete list)

Relatively small set of reserved words which must be memorized.

Predefined identifiers (attributes) may be used as regular identifiers.
PREDEFINED TYPES

INTEGER
FLOAT
BOOLEAN
CHARACTER

Part of pre-defined environment
Not reserved words

PREDEFINED ATTRIBUTES

-- declaration from example
NUMBER_OF_ITEMS :INTEGER range 1..INTEGER'LAST

INTEGER is a predefined type

LAST is a predefined attribute which returns the maximum
value of any scalar type

T'FIRST returns the minimum value of the type T
T'LAST returns the maximum value of the type T
NUMBERS

Integer literals

2500
2.500
25.00
25E2

2#1001_1100_0100#
2#100_111_000_100#

8#4704#
16#9C4#

Different representations of same value

Based integers can be represented with any base from 2 to 16

Real literals

12.75
1275.0E-2
0.1275e2

2#1100.11#
2#110011.0#e-2
2#0.110011#E4

8#14.6#
8#146.0#e1

Different representations of same value
STRINGS

"MAXIMUM IS" -- a string is an array of characters

"/"  -- a string of length one

"HE SAID ""NO""."" -- included string bracket must be
                  -- written twice

"THIS IS "&
"A STRING" -- concatenation used to represent
              -- strings which are longer than
              -- one line

""  -- a one-character string representing
      -- the double quote

""  -- represents an empty string
DELIMITERS

Special characters

+ - / *
, : ; . , 
< = >
( )
| & # _ &

Compound symbols

:= replacement
.. range definition
** exponentiation operation
>= <= /= relational operators
<< >> identifies labels which are objects of GOTO's
=> indicates relationship between a name and a value, action, or declaration
<> stands for unspecified range
COMMENTS

-- This program reads a list of integers

-- A comment starts with a double hyphen
-- and is terminated by the end of the line

begin    -- Body of sort

------------------ the first two hyphens
------------------ start the comment
OBJECT DECLARATIONS

ITEM : INTEGER;

identifier_list : type_mark;

identifier_list : type_mark constraint;

NUMBER_OF_ITEMS : INTEGER range 1..INTEGER'LAST;

Initialization -
identifier_list : type_mark := expression;

COL_NUM, ROW_NUM : INTEGER := 0;
READY, BUSY, RUN : BOOLEAN := FALSE;
RANGE CONSTRAINT

NUMBER_OF_ITEMS : INTEGER
    range 1..INTEGER'LAST;

Form:
    simple_expression .. simple_expression

L .. R describes values from L to R inclusive
L > R indicates empty range
type of range constraint is type of expression
STATEMENTS

ASSIGNMENT

IF

LOOP

SUBPROGRAM CALL
ASSIGNMENT STATEMENT

variable := expression;

   A     A
   |     |
   |___same type___|

MAXIMUM := ITEM;

SUM := SUM + ITEM;

-- compile time checking

-- No automatic conversion

-- across replacement operator
IF STATEMENT

if condition then
sequence_of_statements
end if;

Example

if MONTH = 12 and DAY = 31 then
    MONTH := 1;
    DAY := 1;
    YEAR := YEAR + 1;
end if;
if condition then

sequence_of_statements

elsif condition then

sequence_of_statements

end if;

else

sequence_of_statements

optional

zero or more times
if DAY = DAYS_IN_MONTH then
    DAY := 1;
    if MONTH = 12 then
        MONTH := 1;
        YEAR := YEAR + 1;
    else
        MONTH := MONTH + 1;
    end if;
else
    DAY := DAY + 1;
end if;
DISCRIMINANT := B * B - 4.0 * A * C;
if DISCRIMINANT < 0.0 then
    PUT (" NO REAL ROOTS ");
elsif ABS( DISCRIMINANT ) < 1.0e-8 then
    PUT ( " EQUAL REAL ROOTS ");
    ROOTS := -B/2.0 * A;
    PUT (ROOTS);
else
    PUT (" DISTINCT REAL ROOTS ");
    ...
end if;
LOOP STATEMENT

loop_parameter discrete_range
for N in 2..NUMBER loop
    sequence_of_statements
end loop;

1. The loop parameter is implicitly declared as a local identifier; it (logically) exists only during the execution of the loop statement.

2. The loop parameter acts as a constant; it cannot be altered by the sequence_of_statements.

3. The loop parameter has no value outside the loop.

4. The discrete_range is evaluated only once, before the execution of the loop statement.

5. On successive iterations, the loop parameter is successively assigned values in increasing order from the specified range when in is used. If reserved word reverse is used, values are assigned in decreasing order.
OTHER LOOP EXAMPLES

for N in reverse 1..80 loop
    sequence_of_statments
end loop;

while condition loop
    sequence_of_statments
end loop;
LOOP STATEMENT

Composed of

iteration_specification (optional)

basic_loop

iteration_specification -

while condition

for loop_parameter in discrete_range

for loop_parameter in reverse discrete_range

basic loop -

loop

sequence_of_statements

end loop;
Labeled Loops

SEARCH:

loop
  ...
  ...
end loop SEARCH;

SUMMATION:

for I in 1..N loop
  ...
  ...
end loop SUMMATION;

Compiler will check labels for proper nesting.
SUMMARY

Program Structure
Lexical Units
Declarations
Basic Statements
EXAMPLE II

PROCEDURES AND FUNCTIONS
OBJECTIVES

Procedures and functions
declaration
parameter mode

Blocks

Visibility

Type declarations

Statements

Type equivalence

Operators and operands
type FLOAT_ARRAY is array (INTEGER range <>) of FLOAT;

function AVERAGE (V : in FLOAT_ARRAY) return FLOAT is

    SUM : FLOAT := 0.0;

begin

    for I in V'FIRST..V'LAST loop
        SUM := SUM + V(I);
    end loop;

    return SUM / FLOAT(V'LENGTH);

end AVERAGE;
with MATH_LIB;

procedure STATISTICS (V : in FLOAT_ARRAY;
                      AVG, STD_DEV : out FLOAT) is

  SUM : FLOAT := 0.0;

begin
  AVG := AVERAGE(V);

  for I in V'FIRST..V'LAST loop
    SUM := SUM + (AVG - V(I))**2;
  end loop;

  STD_DEV := MATH_LIB.SQRT(SUM / FLOAT(V'LENGTH));

end STATISTICS;
TYPES and DECLARATIONS

A type characterizes a set of values and a set of operations applicable to those values.

Type declaration

specification of some attributes

association of a name with the attributes

Data object declaration

associates type (attributes) with a name

creates an object of that type

associates the object with the name

Subprogram declaration

associates a block of code with a name

specifies parameters

names, modes, types and order

specify return type (functions)
ARRAY TYPE DEFINITION

name of user-defined type

V

type of index

V

type FLOAT_ARRAY is array (INTEGER range <>)

do FLOAT

A

type of each component
SUBPROGRAMS

Procedures and Functions

subprogram_specification is

declarative_part

begin

sequence_of_statements

end ;
FUNCTIONS

Subprogram specification -

custom AVERAGE ( V : in FLOAT_ARRAY ) return FLOAT

function AVERAGE -- nature and name
   -- of subprogram
   (V : in FLOAT_ARRAY) -- parameter list
      -- (optional)
return FLOAT -- type of object to
      -- be returned
PARAMETER MODES

(V : in FLOAT_ARRAY)

for "in" parameters -

the parameter acts as
a local constant whose
value is provided by
the corresponding actual
parameter

(V : FLOAT_ARRAY) is equivalent to (V : in FLOAT_ARRAY)
function AVERAGE (V : FLOAT_ARRAY) return FLOAT is

    SUM : FLOAT := 0.0;

begin

    for I in V'FIRST..V'LAST loop
        SUM := SUM + V(I);
    end loop;

    return SUM / FLOAT(V'LENGTH);

end AVERAGE;

FIRST, LAST, and LENGTH are predefined attributes

For the array object V,

V'FIRST       lower bound of index of V
V'LAST         upper bound of index of V
V'LENGTH       number of components of V

II.190
PROCEDURES

Subprogram specification -

procedure STATISTICS

(V : in FLOAT_ARRAY;

AVG, STD_DEV : out FLOAT);

for "out" parameters -

the parameter acts as
a local variable whose
value is assigned to the
corresponding actual
parameter at the time
of normal exit
with TEXT_IO, MATH_LIB;

procedure ANALYSIS is

  use TEXT_IO;

  type FLOAT_ARRAY is array (INTEGER range <>) of FLOAT;

  SIZE : NATURAL;

  function AVERAGE (...) is
    ...
  end AVERAGE;

  procedure STATISTICS (...) is
    ...
  end STATISTICS;

begin

  GET(SIZE);

  declare

    RATE : FLOAT_ARRAY(1..SIZE);
    AVERAGE_RATE,
    STD_DEV_RATE : FLOAT;

  begin

    for I in 1..RATE'LAST loop
      GET (RATE(I));
    end loop;

    STATISTICS (RATE, AVERAGE_RATE, STD_DEV_RATE);
      -- use of AVERAGE_RATE and STD_DEV_RATE
    -- in this code

  end;

  -- Variables in block no longer visible

end ANALYSIS;
BLOCK

declare

  declarative_part

begin

  sequence_of_statements

end;

Execution of block results in elaboration of its declarative part followed by execution of the sequence of statements.
function F is
begin
end F;

procedure P is
begin
end P;

declare
begin
end;
SUBTYPES

SIZE : NATURAL;

NATURAL is a predefined identifier

subtype NATURAL is INTEGER
range 1..INTEGER'LAST;

where LAST is a predefined attribute

If T represents a scalar type,
T'LAST returns the maximum value in the range of T.
T'FIRST returns the minimum value in the range of T.
procedure SORT (V : in out FLOAT_ARRAY) is

LAST : INTEGER := V'LAST - 1;
CHANGED : BOOLEAN;
---------------------------------
procedure SWAP (INDEX : in INTEGER) is

TEMP : FLOAT := V(INDEX);

begin -- SWAP
    V(INDEX) := V(INDEX + 1);
    V(INDEX + 1) := TEMP;
end SWAP;
---------------------------------
begin -- SORT
loop
    CHANGED := FALSE;
    for I in V'FIRST..LAST loop
        if V(I+1) < V(I) then
            SWAP(I);
            CHANGED := TRUE;
        end if;
    end loop;
    exit when LAST <= V'FIRST or not CHANGED;
    LAST := LAST - 1;
end loop;
end SORT;
procedure SORT (V : in out FLOAT_ARRAY)

for "in out" parameters -

parameter acts as a local variable and permits access and assignment to the corresponding actual parameter.
procedure SORT ... is

begin  -- body of SORT

end SORT ;
procedure SORT . . . is

LAST : INTEGER := V'LAST - 1;
CHANGED : BOOLEAN;

begin    -- body of SORT

sequence_of_statements

end SORT;
NESTED PROCEDURES

procedure SORT ... is

LAST : INTEGER := V'LAST - 1;
CHANGED : BOOLEAN;
procedure SWAP ... is

end SWAP;

begin -- body of SORT

sequence_of_statements

end SORT ;
procedure SORT (V : in out FLOAT_ARRAY) is

LAST : INTEGER := V'LAST - 1;
CHANGED : BOOLEAN;

procedure SWAP (INDEX : in INTEGER) is

TEMP : FLOAT := V(INDEX);

begin

V(INDEX) := V(INDEX + 1);
V(INDEX + 1) := TEMP;

end SWAP;

begin -- body of SORT

...

end SORT;
procedure OUTER is
    A : BOOLEAN;
    B : BOOLEAN;

    procedure INNER is
        B : BOOLEAN; -- Redefinition hides
        C : BOOLEAN; -- outer B

        begin
            -- Outer A, inner B and C
            -- are directly visible
            -- Outer B can be made visible
            -- by a selected component,
            -- that is, OUTER.B

            ...
        end INNER;

    begin
        -- Outer A and B are directly visible
        -- Inner B and C are not visible

        ...
    end OUTER;
NESTING OF STATEMENTS

begin -- body of SORT

loop

assignment;

for ... loop

if ... then

assignment;

assignment;

end if;

end loop;

exit when ... ;

assignment;

end loop;

end SORT;
LOOP & EXIT STATEMENTS

loop

....

exit when condition;

....

end loop;

exit statement causes explicit termination of enclosing loops

unless ....
REPLACE:

loop

...  
SEARCH:
loop

...  
exit REPLACE when C_ONE ;
...  
exit when C_TWO ;
...  
end loop SEARCH;

...  
end loop REPLACE;
TYPE EQUIVALENCE

type ELEMENT is range 0..K;

A : array (1..N) of 0..K;
B : array (1..N) of 0..K;
C : array (1..N) of ELEMENT;
D : array (1..N) of ELEMENT;

A, B, C, and D are each considered to be of different and distinct types even though the types are textually identical. Thus, the assignment statements

   A := B;
   B := C;

are not allowed.

The assignment

   C(I) := D(I);

is acceptable since the variable and the expression are of the same type (ELEMENT), whereas

   C(I) := B(I)

is not allowed.
A, B : array (1..N) of 0..K;

A and B are objects of the same type

type VECTOR is array (1..N) of 0..K;
C : VECTOR;
D : VECTOR;

C and D are objects of the same type

Whereas A := B and C := D are valid, A := C is not valid.
Different from constraints

I, J : INTEGER range 1..10;
K : INTEGER range 1..20;

I, J and K are all of the same type (i.e., INTEGER)

I := J;  -- identical ranges
K := J;  -- compatible ranges
J := K;  -- can only be checked
          -- during execution

K := 15;
J := K;  -- raise the
          -- RANGE_ERROR exception
TYPES

Scalar types values have no components; includes enumeration, integer, and real types
integer and real called numeric types

Composite types values consist of several component types; includes arrays and records

Access types value provides access to other objects

Scalar

Real

| Discrete |

| ENUMERATION |

| INCLUDES CHARACTER and BOOLEAN |

FLOAT fixed point INTEGER
### Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>and, or, xor, not</td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td></td>
<td>one dimensional array of BOOLEAN components</td>
<td>same array type</td>
</tr>
</tbody>
</table>

**Example:**

```plaintext
type BIT_VECTOR is array (1..32) of BOOLEAN;
A, B : BIT_VECTOR;

Valid expressions:
A and B
A(1..8) or B(1..8)
A(2..5) xor B(29..32)
```
# RELATIONAL OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>any type</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>one dimensional array with components of a discrete type</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

Example:

```plaintext
S, T : array (1..N) of INTEGER;
...

EQUAL := TRUE;
for I in 1..N loop
  if S(I) = T(I) then
    EQUAL := FALSE;
    exit;
  end if;
end loop;
```

can be written as

```plaintext
EQUAL := S = T;
```

Can be extended to multidimensional arrays.
ARITHMETIC OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Left Operand Type</th>
<th>Right Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>integer</td>
<td></td>
<td>same integer type</td>
</tr>
<tr>
<td>-</td>
<td>real</td>
<td></td>
<td>same real type</td>
</tr>
<tr>
<td>*</td>
<td>integer</td>
<td></td>
<td>same integer type</td>
</tr>
<tr>
<td>.</td>
<td>floating</td>
<td></td>
<td>same floating point type</td>
</tr>
<tr>
<td>mod</td>
<td>integer</td>
<td></td>
<td>same integer type</td>
</tr>
<tr>
<td>rem</td>
<td>integer</td>
<td></td>
<td>same integer type</td>
</tr>
</tbody>
</table>

Operator          | Left Operand Type | Right Operand Type | Result Type |
---               |------------------|-------------------|-------------|
**                | integer          | positive integer  | integer     |
                | floating         | integer           | floating    |

TYPE CONVERSIONS

Explicit type conversions allowed between closely related types.

Numeric type conversions:

- REAL( integer expression ) -- value is converted to floating point
- INTEGER ( 1.6 ) = 2 -- conversion of real to integer
- INTEGER (-0.4 ) = 0 -- involves rounding
PRECEDENCE

(lowest) logical and or xor
relational = = <= < > >=
adding + - &
umary + - not
multiplying * mod rem
(Highest) exponentiating **

All operands are evaluated (in an undefined order) before evaluation of the corresponding operator.

Therefore, the expression
A and B or C
requires parentheses; that is
(A and B) or C
or
A and (B or C)

The expressions
A and B and C
and
A or B or C
do not require parentheses.

Short circuit control forms (and then and or else) have same precedence as logical operators.

Membership tests (in and not in) have same precedence as relational operators.
FLOATING POINT TYPE

User defined floating point type:

```
type identifier is floating_point_constraint;
```

where floating_point_constraint is

```
digits P or
digits P range L .. R
```

where D is the required number of digits.

Floating_point_constraint specifies a minimum requirement.

EXAMPLES:

```
type COEFFICIENTS is digits 10 range -1.0 .. 1.0;
type REAL is digits 8;
```

package STANDARD is

```
    type INTEGER is range implementation_defined;
type SHORT INTEGER is range implementation_defined;
type LONG INTEGER is range implementation_defined;
```

```
type FLOAT is digits implementation_defined
    range implementation_defined;
type SHORT FLOAT is digits implementation_defined
    range implementation_defined;
type LONG FLOAT is digits implementation_defined
    range implementation_defined;
```

II.430
FIXED POINT TYPES

EXAMPLE:

    type F is delta 0.01 range -100.0 .. 100.0;

where "delta" of fixed point type specifies the absolute value of the error bound.

If representation uses power of two, 14 bits are required for the magnitude, i.e.,

    64 32 16 8 4 2 1 1/2 1/4 1/8 1/16 1/32 1/64 1/128
\binary point

The error is 1/128 = 0.000_000_1 (base 2) = 0.0078125 < 0.01
SUMMARY

Procedures and functions
declarations
parameter mode

Blocks

Visibility

Type declarations

Statements

Type equivalence

Operators and operands
EXAMPLE III

RECORD HANDLING
OBJECTIVES

Packages

Records and record aggregates

Case statement

Input - Output

Program Structure

Visibility

Separate Compilation
Example III

procedure PROCESS_RECORDS is

package RECORD_HANDLER is
  -- specifications
end RECORD_HANDLER;

use RECORD_HANDLER;
ITEM : ITEM_RECORD; -- defined in RECORD_HANDLER
NO_MORE_RECORDS : BOOLEAN := FALSE;

package body RECORD_HANDLER is
  -- implementation
end RECORD_HANDLER;

begin
  OPEN_FILES;
  loop
    GET_VALID_RECORD (ITEM, NO_MORE_RECORDS);
    exit when NO_MORE_RECORDS;
    WRITE_RECORD (ITEM);
  end loop;
  CLOSE_FILES;
end PROCESS_RECORDS;

-- This specification appears inside of PROCESS_RECORDS, as is
-- indicated above.

package RECORD_HANDLER is
  type ITEM_RECORD is
    record
      ITEM_CODE : record
        PREFIX : STRING(1..2);
        NUMBER : range 0..9_999;
        SUFFIX : CHARACTER;
      end record;
      DESCRIPTION : STRING(1..30);
      SOURCE : range 0..999_999;
    end record;
  procedure OPEN_FILES; 
  procedure CLOSE_FILES;
  procedure GET_VALID_RECORD (REC : out ITEM_RECORD;
                              END_OF_DATA : out BOOLEAN);
  procedure WRITE_RECORD (REC : in ITEM_RECORD);
end RECORD_HANDLER;

III.120
-- This implementation of RECORD_HANDLER is similarly meant to
-- appear within PROCESS_RECORD.

with (TEXT_IO);
package body RECORD_HANDLER is
  use TEXT_IO;
  subtype RECORD_STRING is STRING (1..43);
package RECORD_IO is new INPUT_OUTPUT (ITEM_RECORD);
  IMMEDIATE, DEFERRED : RECORD_IO_OUT_FILE

procedure OPEN_FILES is
  use RECORD_IO;
begin
  CREATE (IMMEDIATE, "external file name");
  CREATE (DEFERRED, "external file name");
end OPEN_FILES;

procedure CLOSE_FILES is
  use RECORD_IO;
begin
  CLOSE (IMMEDIATE);
  CLOSE (DEFERRED);
end CLOSE_FILES;

procedure GET_NEXT_RECORD (REC : out RECORD_STRING;
  VALID_LENGTH,
  ENDOF_DATA : out BOOLEAN) is
  I : NATURAL;
begin
  if CHARACTER IO.END_OF_FILE then
    ENDOF_DATA := TRUE;
  else
    ENDOF_DATA := FALSE;
    I := 0;
    while not END OF LINE and I < 43 loop
      I := I + 1;
      GET (REC(I));
    end loop;
    VALID_LENGTH := I = 43 and END_OF_LINE;
    if not END_OF_LINE then
      SKIP_LINE;
      -- advances input to beginning
      -- of next line
    end if;
  end if;
end GET_NEXT_RECORD;
function VALID_RECORD (REC : in RECORD_STRING) return BOOLEAN is
begin
  function LETTERS (S : STRING) return BOOLEAN is
  begin
    for C in S'FIRST..S'LAST loop
      if S(C) not in 'A'..'Z' and S(C) not in 'a'..'z' then return FALSE; end if;
    end loop;
    return TRUE;
  end LETTERS;

  function NUMERALS (S : STRING) return BOOLEAN is
  begin
    for C in S'FIRST..S'LAST loop
      if S(C) not in '0'..'9' then return FALSE; end if;
    end loop;
    return TRUE;
  end NUMERALS;

begin -- body of VALID_RECORD
  if LETTERS (REC(l..3)) and then NUMERALS (REC(3..6))
    and then (REC(7) = 'N' or REC(7) = 'L' or REC(7) = 'X')
    and then NUMERALS (REC(38..43)) then return TRUE
  else return FALSE end if;
end VALID_RECORD;

procedure WRITE_RECORD (REC : in ITEM_RECORD) is
begin
  case REC.ITEM_CODE.SUFFIX of
  when 'N' => WRITE (IMMEDIATE, REC);
  when 'X' | 'L' => WRITE (DEFERRED, REC);
  others => null;
  end case;
end WRITE_RECORD;

procedure WRITE_ERROR (REC : in RECORD_STRING) is
begin
  PUT("INVALID DATA: " & REC);
  NEW_LINE;
end WRITE_ERROR;
function CONVERT (R : RECORD_STRING) return ITEM_RECORD is

  function STRING_TO_INT (S : STRING) return INTEGER is
    VALUE : INTEGER := 0;
    begin
      for I in S'FIRST..S'LAST loop
        VALUE := 10 * VALUE + CHARACTER'POS(S(I)) - CHARACTER'POS('0');
      end loop;
      return VALUE;
    end STRING_TO_INT;

  begin -- body of CONVERT
    return (ITEM_CODE => (R(1..2),
                           STRING_TO_INT (R(3..6)),
                           R(7)),
        DESCRIPTION => R(8..37),
        SOURCE => STRING_TO_INT (R(38..43)));
  end CONVERT;

procedure GET_VALID_RECORD (REC : out ITEM_RECORD);
ENDOF : out BOOLEAN) is
  S : RECORD_STRING;
  LENGTH_ERROR : BOOLEAN;
  begin
    loop
      GET_NEXT_RECORD (S , LENGTH_ERROR, ENDOF_DATA);
      if ENDOF_DATA then
        return;
      elsif LENGTH_ERROR or else not VALID_RECORD(S) then
        WRITE_ERROR(S);
      else
        REC := CONVERT(S);
        return;
      end if;
    end loop;
  end GET_VALID_RECORD;

end RECORD_HANDLER;
INPUT VALIDATION and FILE SELECTION

FILE OF RECORDS (INPUT)

\[
\begin{array}{c}
\text{FILE HANDLER} \\
\text{file: OUTPUT Invalid records} \\
\text{file: IMMEDIATE} \\
\text{file: DEFERRED}
\end{array}
\]

INPUT: string (array of characters)
OUTPUT: string
IMMEDIATE: file of records
DEFERRED: file of records
## INPUT RECORD FORMAT

(valid records)

<table>
<thead>
<tr>
<th>POSITION</th>
<th>NAME</th>
<th>CONTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-7</td>
<td>ITEMCODE</td>
<td>2 ALPHABETIC CHARACTERS</td>
</tr>
<tr>
<td></td>
<td>- PREFIX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- NUMBER</td>
<td>4 NUMERALS</td>
</tr>
<tr>
<td></td>
<td>- SUFFIX</td>
<td>N, L, or X</td>
</tr>
<tr>
<td>8-37</td>
<td>DESCRIPTION</td>
<td>30 CHARACTERS</td>
</tr>
<tr>
<td>38-43</td>
<td>SOURCE</td>
<td>6 NUMERALS</td>
</tr>
</tbody>
</table>
Input:

```plaintext
subtype RECORD_STRING is STRING (1..43);
REC : RECORD_STRING;
```

Valid Input

Output files IMMEDIATE

and DEFERRED

```plaintext
REC (1..7) ITEMCODE
REC (1..2) PREFIX
REC (3..6) CONVERT NUMBER
REC (7) ---------> SUFFIX
REC (8..37) DESCRIPTION
REC (38..43) SOURCE
```
ARRAY OBJECT -

a set of components in which each component is of the same type

array component is designated by one or more index values

RECORD OBJECT -

a set of components in which the components may be of different types

a record object has named components
**RECORD STRUCTURE**

<table>
<thead>
<tr>
<th>ITEM_CODE</th>
<th>DESCRIPTION</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX</td>
<td>NUMBER</td>
<td>SUFFIX</td>
</tr>
</tbody>
</table>

```plaintext
type ITEM_RECORD is
  record
    ITEM_CODE : record
      PREFIX : STRING(1..2);
      NUMBER : range 0..9_999;
      SUFFIX : CHARACTER;
    end record;
    DESCRIPTION : STRING (1..30);
    SOURCE : range 0..999_999;
  end record;
```
Object declaration:

    REC : ITEM_RECORD;

Reference to the components:

    REC.SOURCE := 475124;
    REC.ITEM_CODE.PREFIX := "PS";
    case REC.ITEM_CODE.SUFFIX is
PROGRAM DESIGN

initialize

loop
    get valid record
    exit when no more records
    write to selected file
end loop

end loop

clean up
PROGRAM STRUCTURE

--- OPEN FILES

--- GET NEXT RECORD

GET

-VALID-

RECORD

--- VALID RECORD

--- LETTERS

--- NUMERALS

--- WRITE ERROR

--- CONVERT---STRING TO INT

--- WRITE RECORD

--- CLOSE FILES
PACKAGE SPECIFICATION

package RECORD_HANDLER is
    type ITEM_RECORD is
        record
            ITEM_CODE : record
                PREFIX : STRING(1..2);
                NUMBER : range 0..9_999;
                SUFFIX : CHARACTER;
            end record;
            DESCRIPTION : STRING(1..30);
            SOURCE : range 0..999_999;
        end record;
    procedure OPEN_FILES;
    procedure CLOSE_FILES;
    procedure GET_VALID_RECORD (REC : out ITEM_RECORD;
                                 END_OF_DATA : out BOOLEAN);
    procedure WRITE_RECORD (REC : in ITEM_RECORD);
end RECORD_HANDLER;
procedure PROCESS_RECORDS is

package RECORD_HANDLER is
    -- specifications
end RECORD_HANDLER;

use RECORD_HANDLER;
ITEM : ITEM_RECORD; -- defined in RECORD_HANDLER
NO_MORE_RECORDS : BOOLEAN := FALSE;

package body RECORD_HANDLER is
    -- implementation
end RECORD_HANDLER;

begin
    OPEN_FILES;
    loop
        GET_VALID_RECORD (ITEM, NO_MORE_RECORDS);
        exit when NO_MORE_RECORDS;
        WRITE_RECORD (ITEM);
    end loop;
    CLOSE_FILES;
end PROCESS_RECORDS;
Outline of RECORD_HANDLER

with TEXT_IO;

package body RECORD_HANDLER is

  use TEXT_IO;
  subtype RECORD_STRING is STRING (1..43);
  package RECORD_IO is new INPUT_OUTPUT
    (ITEM_RECORD);
  IMMEDIATE, DEFERRED : RECORD_IO.OUT_FILE;

  procedure OPEN_FILES is
    ...
  end OPEN_FILES;

  procedure CLOSE_FILES is
    ...
  end CLOSE_FILES;

  procedure GET_NEXT_RECORD (REC : out RECORD_STRING;
    VALID_LENGTH,
    END_OF_DATA : out BOOLEAN) is
    ...
  end GET_NEXT_RECORD;

  function VALID_RECORD (REC : in RECORD_STRING)
    return BOOLEAN is
    function LETTERS (S : STRING) return BOOLEAN is
      ...
    end LETTERS;
    function NUMERALS (S : STRING) return BOOLEAN is
      ...
    end NUMERALS;
    ...
  end VALID_RECORD;
procedure WRITE_RECORD (REC : in ITEM_RECORD) is
...
end WRITE_RECORD;

procedure WRITE_ERROR (REC : in RECORD_STRING) is
...
end WRITE_ERROR;

function CONVERT (R : RECORD_STRING) return ITEM_RECORD is
...
function STRING_TO_INT (S : STRING) return INTEGER is
...
end STRING_TO_INT;
...
end CONVERT;

procedure GET_VALID_RECORD (REC : out ITEM_RECORD;
END_OF_DATA : out BOOLEAN) is
...
end GET_VALID_RECORD;

end RECORD_HANDLER;
procedure GET_VALID_RECORD (REC : out ITEM_RECORD;
   END_OF_DATA : out BOOLEAN) is

   S : RECORD_STRING;
   LENGTH_ERROR : BOOLEAN;

begin

   loop

      GET_NEXT_RECORD (S, LENGTH_ERROR, END_OF_DATA);

      if END_OF_DATA then
         return;
      elsif LENGTH_ERROR or else not VALID_RECORD(S) then
         WRITE_ERROR(S);
      else
         REC := CONVERT(S);
         return;
      end if;

   end loop;

end GET_VALID_RECORD;
SHORT CIRCUIT CONDITION

or else

expression-1 or expression-2

expression-2 will be evaluated even if expression-1 is true

expression-1 or else expression-2

if expression-1 is true, expression-2 is not evaluated

A or else B or else C

evaluation of expressions (A,B,C) proceeds in textual order

evaluation stops as soon as an expression evaluates to true
procedure GET_NEXT_RECORD (REC : out RECORD_STRING;
            VALID_LENGTH,
            END_OF_DATA : out BOOLEAN) is

    I : NATURAL;

begin

    if CHARACTER_IO.END_OF_FILE then
        END_OF_DATA := TRUE
    else
        END_OF_DATA := FALSE;
        I := 0;
        while not END_OF_LINE and I < 43 loop
            I := I + 1;
            GET (REC(I));
        end loop;
        VALID_LENGTH := I = 43 and END_OF_LINE;
    if not END_OF_LINE then
        SKIP_LINE;
        -- advances input to beginning
        -- of next line
    end if;
end if;

end GET_NEXT_RECORD;
function VALID_RECORD ... is

function LETTERS ... is
begin
...
end LETTERS;

function NUMERALS ... is
begin
...
end NUMERALS;

begin  -- body of VALID_RECORD
...
end VALID_RECORD
VALID_RECORD

function VALID_RECORD (REC : in RECORD_STRING) return BOOLEAN is

function LETTERS (S : STRING) return BOOLEAN is
begin
  for C in S'FIRST..S'LAST loop
    if S(C) not in 'A'..'Z' and S(C) not in 'a'..'z'
      then return FALSE;
    end if;
  end loop;
  return TRUE;
end LETTERS;
MEMBERSHIP OPERATOR

if S(C) not in 'A'..'Z' and
   S(C) not in 'a'..'z' then
   return FALSE;

'in' and 'not in' are membership
operators

test for membership of a value
of any type within a corresponding
range, subtype, or constraint

returns boolean value

same precedence as relational
operators
function NUMERALS (S : STRING)
  return BOOLEAN is
begin
  for C in S'FIRST..S'LAST loop
    if S(C) not in '0'..'9' then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end NUMERALS;
SHORT CIRCUIT CONDITION

begin -- body of VALID_RECORD

if LETTERS (REC(1..2)) and then NUMERALS (REC(3..6))
    and then (REC(7) = 'N' or REC(7) = 'L' or REC(7) = 'X')
    and then NUMERALS (REC(38..43)) then
    return TRUE;
else
    return FALSE;
end if;
end VALID_RECORD;

if C1 and then C2 and then C3 then
...

is equivalent to

if C1 then
    if C2 then
        if C3 then
            ...

III.350
<table>
<thead>
<tr>
<th>All Character (STRING)</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(1..2)</td>
<td>PREFIX</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>R(3..6) =&gt; STRING_TO_INT</td>
<td>NUMBER</td>
<td>1..9_999</td>
</tr>
<tr>
<td>R(7)</td>
<td>SUFFIX</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>R(8..37)</td>
<td>DESCRIPTION</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>R(38..43) =&gt; STRING_TO_INT</td>
<td>SOURCE</td>
<td>1..999_999</td>
</tr>
</tbody>
</table>
PREDEFINED ATTRIBUTE

POS

T'POS(X) gives the ordinal position
of the value X in the
discrete type T

T'POS(T'FIRST) = 0

type CHARACTER is

(nul, soh, stx, etx, ...
'0', '1', '2', ..., '9', ...
'A', 'B', 'C', ..., 'Z', ...
'a', 'b', 'c', ..., 'z', ...);

Standard ASCII character set

CHARACTER'POS(NUL) = 0
CHARACTER'POS(CHARACTER'LAST) = 127

CHARACTER'POS('3') ≠ 3

CHARACTER'POS('3')
- CHARACTER'POS('0') = 3
function DEC (C : CHARACTER) return INTEGER is
    BASE : constant INTEGER := CHARACTER'POS('0');

    begin
        return CHARACTER'POS(C) - BASE;
    end DEC;

    DEC('4') = 4  
    DEC('7') = 7

S := "475"

N := 0;
for I in S'FIRST..S'LAST loop
    N := N * 10 + DEC(S(I));
end loop;
function STRING_TO_INT ( S : STRING )

    return INTEGER is

    VALUE : INTEGER := 0;

    begin

    for I in S'FIRST .. S'LAST loop

        VALUE := VALUE * 10
          + CHARACTER'POS(S(I))
          - CHARACTER'POS('0');

    end loop;

    return VALUE;

    end STRING_TO_INT;
function STRING_TO_INT ( S : STRING )
    return INTEGER is ...

STRING_TO_INT ( "451" ) = 451
              451
               \\        \v
STRING   INTEGER

-- declaration
PHONE_NUMBER : STRING (1..10);

-- assignment
PHONE_NUMBER := "4048943181";

-- declaration
AREA_CODE, EXTENSION : INTEGER;

-- assignment
AREA_CODE :=
    STRING_TO_INT( PHONE_NUMBER (1..3) );
    -- sets AREA_CODE to 404

EXTENSION :=
    STRING_TO_INT( PHONE_NUMBER (7..10) )
    -- sets EXTENSION to 3181
-- declarations
PHONE_NUMBER : STRING (1..10);
AREA_CODE : STRING (1..3);
EXTENSION : STRING (1..4);

-- assignments
PHONE_NUMBER := "4048943181";

AREA_CODE := PHONE_NUMBER (1..3);
EXTENSION := PHONE_NUMBER (7..10);

PHONE_NUMBER (7..10) := "1815";

PHONE_NUMBER (4..6) :=
    PHONE_NUMBER (1..3);

PHONE_NUMBER (1..5) :=
    PHONE_NUMBER (3..7);
function CONVERT ( R : RECORD_STRING )
return ITEM_RECORD is

function STRING_TO_INT ...
...
end STRING_TO_INT;

begin

return ( ITEM_CODE => ( R(1..2),
STRING_TO_INT ( R(3..6) ),
R (7) ),

DESCRIPTION => R (8..37),

QUANTITY =>
STRING_TO_INT ( R(38..43) ) );

end CONVERT;
A record aggregate denotes a value constructed from component values.

NEW_ITEM : ITEM_CODE;  -- object declaration

NEW_ITEM := ( "CT", 2493, 'N' )  -- assignment

NEW_ITEM := ( NUMBER => 2493, PREFIX => "CT", SUFFIX => 'N' )  -- named components
RECORD AGGREGATE

-- named component

(ITEM_CODE =>

    -- positional
    ( R(1..2),
        -- PREFIX
        STRING_TO_INT( R(3..6)),
        -- NUMBER
        R(7)),
        -- SUFFIX

-- named component

DESCRIPTION => R(8..37),

    array slice

-- named component

SOURCE => STRING_TO_INT( R(38..43) )

    array slice
The package TEXT_IO contains the definition of all the text input-output primitives.

It contains the specifications:

- procedure GET(ITEM : out CHARACTER);
- procedure PUT(ITEM : in CHARACTER);
- procedure PUT(ITEM : in STRING);
WRITE_ERROR and WRITE_RECORD

procedure WRITE_ERROR (REC : in RECORD_STRING) is
begin
  PUT("INVALID DATA: " & REC);
  NEW_LINE;
end WRITE_ERROR;

procedure WRITE_RECORD (REC : in ITEM_RECORD) is
use RECORD_IO;
begin
  case REC.ITEM_CODE.SUFFIX is
    when 'N' => WRITE (IMMEDIATE, REC);
    when 'X' => WRITE (DEFERRED, REC);
    others => null;
  end case;
end WRITE_RECORD;
TEXT FILES

All characters occupy exactly one column.

Characters of a file are considered to form a sequence of lines.

Layout control

LINE - returns current line number
COL - returns current column number
END_OF_LINE - returns TRUE if there is no character left on the current input line (defined for IN_FILE only)
SKIP_LINE - advances the input to the beginning of the next line (defined only for IN_FILE)
NEW_LINE - terminates current output line (defined only for OUT_FILE)
SET_COL - sets the current column number
SET_LINE_LENGTH - sets the line length
LINE_LENGTH - returns current line length
FILE OF RECORDS

A file is associated with an ordered collection of elements, all of the same type.

Let $E_t$ denote an element of type $T$.

\[ E_t \quad E_t \quad E_t \quad \ldots \quad E_t \quad E_t \]

In this example, each $E_t$ is a record

\[ E_t = \begin{array}{c|c|c|c|c|c} \hline \text{ITEM_CODE} & \hline \text{PREFIX} & \text{NUMBER} & \text{SUFFIX} & \text{DESCRIPTION} & \text{QUANTITY} \end{array} \]
package RECORD_IO is new INPUT_OUTPUT (ITEM_RECORD);

INPUT_OUTPUT is a standard generic package which provides the calling conventions for operations such as OPEN, CLOSE, READ, and WRITE.

generic (type ELEMENT_TYPE)
package INPUT_OUTPUT is

   ...
   procedure WRITE (FILE : in OUT_FILE;
               ITEM : in
               ELEMENT_TYPE);
   ...

A generic package is a model which can be parameterized.
package RECORD_10 is new
  INPUT_OUTPUT ( ITEM_RECORD );

  parameter

  generic instantiation

  obtains a copy (instance)
  of the model with actual
  parameter ITEM_RECORD
  substituted for the
  generic formal parameter
  ELEMENT_TYPE.
IMMEDIATE, DEFERRED : RECORD_IO.OUT_FILE;

OUT_FILE is a file type with write-only access

it is declared in the package INPUT_OUTPUT

it is instantiated within RECORD_IO
OPEN_FILES and CLOSE_FILES

The generic standard package INPUT_OUTPUT contains the specifications

procedure CREATE(FILE : in out OUT_FILE;
    NAME : in STRING);

which establishes a new external file associates the given file with it; this association "opens" the file, and

procedure CLOSE(FILE : in out OUT_FILE);

which breaks the association.

procedure OPEN_FILES is
    use RECORD_IO;
    begin
        CREATE (IMMEDIATE, "external file name");
        CREATE (DEFERRED, "external file name");
    end OPEN_FILES;

procedure CLOSE_FILES is
    use RECORD_IO;
    begin
        CLOSE (IMMEDIATE);
        CLOSE (DEFERRED);
    end CLOSE_FILES;
PROGRAM STRUCTURE

Packages are a versatile feature used in a number of ways in the construction of Ada programs.

Packages allow for the specification of groups of logically related entities:

- pools of common data and associated type declarations
- groups of related subprograms (either within a single program or as a subprogram library)
- a type declaration along with subprograms to serve as operators on the type (data abstraction)

The separation of a package body from its specification provides an important information hiding capability.
package WORK_DATA is

    type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
    type HOURS is INTEGER range 0..2400;
    type TIME_TABLE is
        array (MON..SUN) of HOURS;

    WORK_HOURS : TIME_TABLE;
    NORMAL_HOURS : constant TIME_TABLE
        := ( MON..THU => 850, FRI => 600,
             SAT | SUN => 0 );

end WORK_DATA;
VISIBILITY

procedure EXAMPLE ...  

    package WORK_DATA is 
    
    end WORK_DATA; 
    
    Identifiers declared within WORK_DATA 
    can be used here, denoted by 
    
    selected components

    Examples of legal references: 
    WORK_DATA.DAY
    WORK_DATA.WORK_HOURS

    end EXAMPLE;

WORK_DATA and its components are not 
visible outside of EXAMPLE.
procedure EXAMPLE...

package WORK_DATA is
  ...
end WORK_DATA;
  ...
procedure P2 ...
  ...
  use WORK_DATA;

  Identifiers declared within WORK_DATA are now directly visible.

  Examples of legal references:
  TIME_TABLE
  NORMA HOURS
  ...
  
end P2;

  The use clause is no longer effective outside of P2, so selected component notation must again be used to reference the objects defined within WORK_DATA.
  ...
end EXAMPLE;

  WORK_DATA and its components are again not visible at this point.
procedure MAIN is

package WORK_DATA is

   NORMAL_HOURS : constant TIME_TABLE
     := (MON..THU => 850, FRI => 600,
         SAT | SUN => 0);
end WORK_DATA;

procedure A is

   use WORK_DATA;

   NORMAL_HOURS : INTEGER;

   -- NORMAL_HOURS refers to the integer;
   -- it cannot be hidden by the
   -- the same identifier declared
   -- in the package.

   -- The use clause makes all identifiers
   -- in the package directly visible
   -- except for the identifier NORMAL_HOURS.

   -- It can only be denoted as a
   -- selected component, that is,
   -- WORK_DATA.NORMAL_HOURS (...)

   end A;

end MAIN;
procedure PROCESS_RECORDS is

    package RECORD_HANDLER is
    end RECORD_HANDLER;

    use RECORD_HANDLER;
    ITEM : ITEM_RECORD;

    package body RECORD_HANDLER is
    end RECORD_HANDLER;

begin

end PROCESS_RECORDS;
procedure PROCESS_RECORDS is

package RECORD_HANDLER is
    -- type & variable declarations
    -- subprogram specifications
end RECORD_HANDLER;

use RECORD_HANDLER;
ITEM : ITEM_RECORD;
    -- variable declaration

package body RECORD_HANDLER is
    -- type & variable declarations
    -- subprogram bodies
end RECORD_HANDLER;

begin

end PROCESS_RECORDS;
procedure PROCESS_RECORDS is

package RECORD_HANDLER is
  -- package specification
  -- visible part
  end RECORD_HANDLER;

use RECORD_HANDLER;
ITEM : ITEM_RECORD;
  -- variable declaration

package body RECORD_HANDLER is
  -- package body
  -- entities not accessible
  -- outside the package
  end RECORD_HANDLER;

begin

end PROCESS_RECORDS;
SEPARATE COMPILATION

PROGRAM – collection of one or more compilation units

COMPILATION UNIT –
  . subprogram body
  . package specification
  . package body

Compilation units of a program are said to belong to a
PROGRAM LIBRARY
The package body is to be compiled separately.

```
procedure PROCESS_RECORDS is

package RECORD_HANDLER is
    -- contains type declarations
    -- and subprogram specifications
end RECORD_HANDLER;

use RECORD_HANDLER;
ITEM : ITEM_RECORD;

package body RECORD_HANDLER
    is separate;
begin
    ...
end PROCESS_RECORDS;

separate ( PROCESS_RECORDS )
with TEXT_IO;
package body RECORD_HANDLER is
    -- local declarations
    -- subprogram bodies
    ...
end RECORD_HANDLER;
```
COMPILATION OF PACKAGE BODY

separate (PROCESS_RECORDS)
with TEXT_IO;
package body RECORD_HANDLER is

-- local declarations and the bodies
-- of the subprograms declared in
-- the specification part are found
-- in the package body

end RECORD_HANDLER;

The with clause indicates that the package TEXT_IO will be used in this package body.

The separate clause says that the specifications for this package can be found in the program unit named PROCESS_RECORDS. Identifiers visible at the point of the separate declaration in PROCESS_RECORDS are also visible in the package body.
SEPARATE COMPILATION

Version 2

Three program units
1. package specification
2. subprogram (program)
3. package body

Each compiled separately.

Package specification must be compiled first.

Procedure and package body may be compiled (and recompiled) in any order.

The package body is no longer within PROCESS_RECORDS, so no separate clause is used.
package RECORD_HANDLER is
   -- type declarations and
   -- subprogram specifications
   ...
end RECORD_HANDLER;

with RECORD_HANDLER;
procedure PROCESS_RECORDS is
   use RECORD_HANDLER;
   ...
begin
   ...
end PROCESS_RECORDS;

with TEXT_IO;
package body RECORD_HANDLER is
   use TEXT_IO;
   -- declaration of entities
   -- not accessible outside
   -- package body and
   -- subprogram bodies
   ...
   function VALID_RECORD ... 
      return BOOLEAN is separate;
   ...
end PROCESS_RECORDS;

separate (RECORD_HANDLER)
function VALID_RECORD ... is
   ...
end VALID_RECORD
Within the body of RECORD_HANDLER, the separate compilation of a subprogram within another program unit is illustrated.

function VALID_RECORD ( REC : in RECORD_STRING )
return BOOLEAN is separate;

The body of this function would be compiled as a fourth compilation unit. It must be compiled after the body of RECORD_HANDLER (and recompiled any time that body is recompiled).

separate (RECORD_HANDLER)
function VALID_RECORD ... is
  :
  :
end VALID_RECORD;
package RECORD | HANDLER is 
| type ITEM | RECORD is 
| record 
| ITEM_CODE : record 
| PREFIX : STRING(l..2); 
| NUMBER : range 0..9_999; 
| SUFFIX : CHARACTER; 
| end record; 
| DESCRIPTION : STRING(l..30); 
| SOURCE : range 0..999_999; 
| end record; 
| procedure OPEN FILES; 
| procedure CLOSE FILES; 
| procedure GET_VALID_RECORD (REC : out ITEM RECORD; 
| END OF DATA : out BOOLEAN); 
| procedure WRITE_RECORD (REC : in ITEM_RECORD); 
| end RECORD HANDLER; 

with RECORD HANDLER; 
procedure PROCESS RECORDS is 

use RECORD HANDLER; 
ITEM : ITEM RECORD; -- defined in RECORD HANDLER 
NO_MORE_RECORDS : BOOLEAN := FALSE; 

begin 
OPEN FILES; 
loop 
GET_VALID_RECORD (ITEM, NO_MORE_RECORDS); 
exi when NO_MORE_RECORDS; 
WRITE_RECORD (ITEM); 
end loop; 
CLOSE FILES; 
end PROCESS_RECORDS;
with TEXT_IO;
package body RECORD_HANDLER is
  use TEXT_IO;
  subtype RECORD_STRING is STRING (1..43);
package RECORD_IO is new INPUT_OUTPUT (ITEM_RECORD);
IMMEDIATE, DEFERRED : RECORD_IO.OUT_FILE;

procedure OPEN_FILES is
  use RECORD_IO;
begin
  CREATE (IMMEDIATE, "external file name");
  CREATE (DEFERRED, "external file name");
end OPEN_FILES;

procedure CLOSE_FILES is
  use RECORD_IO;
begin
  CLOSE (IMMEDIATE);
  CLOSE (DEFERRED);
end CLOSE_FILES;

procedure GET_NEXT_RECORD (REC : out RECORD_STRING;
VALID_LENGTH,
END_OF_DATA : out BOOLEAN) is
  I : NATURAL;
begin
  if CHARACTER_IO.END_OF_FILE then
    END_OF_DATA := TRUE;
  else
    END_OF_DATA := FALSE;
    I := 0;
    while not END_OF_LINE and I < 43 loop
      I := I + 1;
      GET (REC(I));
    end loop;
    VALID_LENGTH := I = 43 and END_LINE;
    if not END_OF_LINE then
      SKIP LINE;
      -- advances input to beginning
      -- of next line
    end if;
  end if;
end GET_NEXT_RECORD;

function VALID_RECORD (REC : in RECORD_STRING)
return BOOLEAN is separate;
procedure WRITE_RECORD (REC : in ITEM_RECORD) is
use RECORD_IO;
begin
  case REC.ITEM CODE.SUFFIX of
    when 'N' => WRITE (IMMEDIATE, REC);
    when 'X' | 'L' => WRITE (DEFERRED, REC);
    others => null;
  end case;
end WRITE_RECORD;

procedure WRITE_ERROR (REC : in RECORD_STRING) is
begin
  PUT("INVALID DATA: ", REC);
  NEW_LINE;
end WRITE_ERROR;

function CONVERT (R : RECORD STRING) return ITEM_RECORD is
  function STRING TO INT (S : STRING) return INTEGER is
    function STRING_TO_INT (S : STRING) return INTEGER is
      VALUE := 0;
      begin
        for I in S'FIRST..S'LAST loop
          VALUE := 10 * VALUE + CHARACTER'POS(S(I)) - CHARACTER'POS ('0');
        end loop;
        return VALUE;
      end STRING_TO_INT;
begin -- body of CONVERT
  return (ITEM_CODE => (R(1..2),
                        STRING_TO_INT (R(3..6)),
                        R(7)),
            DESCRIPTION => R(8..37),
            SOURCE => STRING_TO_INT (R(38..43)));
end CONVERT;

procedure GET_VALID_RECORD (REC : out ITEM_RECORD);  END_OF_DATA : out BOOLEAN) is
begin
  loop
    GET NEXT RECORD (S, LENGTH_ERROR, END_OF_DATA);
    if END_OF_DATA then
      return;
    elsif LENGTH_ERROR or else not VALID_RECORD(S) then
      WRITE_ERROR(S);
    else
      REC := CONVERT(S);
      return;
    end if;
  end loop;
end GET_VALID_RECORD;
end RECORD_HANDLER;
separate (RECORD HANDLER)
function VALID_RECORD (REC : in RECORD_STRING)
    return BOOLEAN is

    function LETTERS (S : STRING) return BOOLEAN is
        begin
            for C in S'FIRST..S'LAST loop
                if S(C) not in 'A'..'Z' and S(C) not in 'a'..'z'
                    then return FALSE;
                end if;
            end loop;
            return TRUE;
        end LETTERS;

    function NUMERALS (S : STRING) return BOOLEAN is
        begin
            for C in S'FIRST..S'LAST loop
                if S(C) not in '0'..'9'
                    then return FALSE;
                end if;
            end loop;
            return TRUE;
        end NUMERALS;

    begin -- body of VALID_RECORD
        if LETTERS (REC(1..2)) and then NUMERALS (REC(3..6))
            and then (REC(7) = 'N' or REC(7) = 'L' or REC(7) = 'X')
            and then NUMERALS (REC(38..43)) then
            return TRUE
        else
            return FALSE
        end if;
    end VALID_RECORD;
SUMMARY

Packages
Records and record aggregates
Case statement
Input-Output
Program Structure
Visibility
Separate Compilation
EXAMPLE IV

ENUMERATION TYPES
OBJECTIVES

Enumeration Types

Array Aggregates

Named Parameter Association
package NAVIGATION is

  type DIRECTION is (NORTH, EAST, SOUTH, WEST);
  type TURN is (LEFT, RIGHT, ABOUT, NONE);

  function TURN_LEFT (D : DIRECTION) return DIRECTION;
  function TURN_RIGHT (D : DIRECTION) return DIRECTION;
  function TURN_ABOUT (D : DIRECTION) return DIRECTION;
  function CHANGE_COURSE (D : DIRECTION; T : TURN) return DIRECTION;
  function MANEUVER (OLD, NEW : DIRECTION) return TURN;

end NAVIGATION;

package body NAVIGATION is

  function TURN_LEFT (D : DIRECTION) return DIRECTION is
    -- declare a local variable to illustrate use
    -- of a single return at the end of the body
    NEW_D : DIRECTION;

    begin
      case D of
        when NORTH => NEW_D := WEST;
        when SOUTH => NEW_D := EAST;
        when EAST => NEW_D := NORTH;
        when WEST => NEW_D := SOUTH;
      end case;

      return NEW_D;
    end TURN_LEFT;
function TURN_RIGHT (D : DIRECTION) return DIRECTION is

-- a return statement will appear in each
-- alternative of the case statement

begin

case D is
  when NORTH => return EAST;
  when SOUTH => return WEST;
  when EAST => return SOUTH;
  when WEST => return NORTH;
end case;

end TURN_RIGHT;

----------------------------------------

function TURN_ABOUT (D : DIRECTION) return DIRECTION is

-- look up answer in a constant array

NEW_D : constant array (DIRECTION) of DIRECTION := (
  NORTH => SOUTH,
  SOUTH => NORTH,
  EAST => WEST,
  WEST => EAST);

begin

  return NEW_D(D);

end TURN_ABOUT;

----------------------------------------

function CHANGE_COURSE (D : DIRECTION; T : TURN) return DIRECTION is

begin

case T is
  when LEFT => return TURN_LEFT(D);
  when RIGHT => return TURN_RIGHT(D);
  when ABOUT => return TURN_ABOUT(D);
  when NONE => return D;
end case;

end CHANGE_COURSE;

----------------------------------------
function MANEUVER ( OLD, NEW : DIRECTION ) return TURN is
begin
    if NEW = TURN LEFT( OLD ) then
        return LEFT;
    elsif NEW = TURN RIGHT( OLD ) then
        return RIGHT;
    elsif NEW = TURN ABOUT( OLD ) then
        return ABOUT;
    else
        return NONE;
    end if;
end MANEUVER;

end NAVIGATION;
package NAVIGATION is

  type DIRECTION is (NORTH, EAST, SOUTH, WEST);
  type TURN is (LEFT, RIGHT, ABOUT, NONE);

  function TURN_LEFT (D : DIRECTION) return DIRECTION;
  function TURN_RIGHT (D : DIRECTION) return DIRECTION;
  function TURN_ABOUT (D : DIRECTION) return DIRECTION;
  function CHANGE_COURSE (D : DIRECTION; T : TURN) return DIRECTION;
  function MANEUVER (OLD, NEW : DIRECTION) return TURN;

end NAVIGATION;
package body NAVIGATION is

  function TURN_LEFT ... is
    ...
  end TURN_LEFT;

  function TURN_RIGHT ... is
    ...
  end TURN_RIGHT;

  function TURN_ABOUT ... is
    ...
  end TURN_ABOUT;

  function CHANGE_COURSE ... is
    ...
  end CHANGE_COURSE;

  function MANEUVER ... is
    ...
  end MANEUVER;

end NAVIGATION;
ENUMERATION TYPES

type DIRECTION is
   (NORTH, EAST, SOUTH, WEST);

OLD_D, NEW_D : DIRECTION;

OLD_D := NORTH;
NEW_D := OLD_D;

Predefined attributes:

DIRECTION'FIRST = NORTH
DIRECTION'LAST = WEST

DIRECTION'SUCC(EAST) = SOUTH
DIRECTION'PRED(WEST) = SOUTH
DIRECTION'POS(SOUTH) = 2

DIRECTION'SUCC(DIRECTION'LAST) -- raise the exception
DIRECTION'PRED(DIRECTION'FIRST) -- OBJECT_ERROR
function TURN_LEFT (D : DIRECTION) return DIRECTION is

   -- declare a local variable to illustrate use
   -- of a single return at the end of the body

   NEW_D : DIRECTION;

begin

   case D is
      when NORTH => NEW_D := WEST;
      when SOUTH => NEW_D := EAST;
      when EAST => NEW_D := NORTH;
      when WEST => NEW_D := SOUTH;
   end case;

   return NEW_D;
end TURN_LEFT;

function TURN_RIGHT (D : DIRECTION) return DIRECTION is

   -- a return statement will appear in each
   -- alternative of the case statement

begin

   case D is
      when NORTH => return EAST;
      when SOUTH => return WEST;
      when EAST => return SOUTH;
      when WEST => return NORTH;
   end case;

end TURN_RIGHT;
The order relations between enumeration values follow the order of listing:

NORTH < EAST < SOUTH < WEST

for D in NORTH .. WEST loop
...
end loop;

for D in DIRECTION'FIRST .. DIRECTION'LAST loop
...
end loop;
Alternate solution to TURN_RIGHT

function TURN_RIGHT (D : DIRECTION) return DIRECTION is
begin
    if D = DIRECTION'LAST then
        return DIRECTION'FIRST;
    else
        return DIRECTION'SUCC(D);
    end if;
end TURN_RIGHT;
function TURN_ABOUT ( D : DIRECTION ) return DIRECTION is

    -- look up answer in a constant array

    NEW_D : constant array ( DIRECTION ) of DIRECTION :=
    ( NORTHERN => SOUTH,
    SOUTH => NORTH,
    EAST => WEST,
    WEST => EAST);

begin

    return NEW_D( D );

end TURN_ABOUT;
ARRAY INDEXED BY
ENUMERATION

function TURN_About (D : DIRECTION)
    return DIRECTION is
    NEW_D : constant array (DIRECTION) of DIRECTION :=
    (NORTH => SOUTH,
    SOUTH => NORTH,
    EAST => WEST,
    WEST => EAST);

    -- NEW_D is a one-dimensional
    -- array with four components

    -- Each element (or component)
    -- may take on one of the
    -- enumerated values of type
    -- DIRECTION

    -- The four elements are
    -- denoted by

    -- NEW_D(NORTH)
    -- NEW_D(EAST)
    -- NEW_D(SOUTH)
    -- NEW_D(WEST)
ARRAY AGGREGATES

NEW_D : constant array (DIRECTION)
of DIRECTION

:= ( NORTH => SOUTH,
SOUTH => NORTH,
EAST => WEST,
WEST => EAST ),

-- NEW_D(NORTH) = SOUTH
-- NEW_D(SOUTH) = NORTH
-- NEW_D(EAST) = WEST
-- NEW_D(WEST) = EAST

begin

return NEW_D (D);

end TURN_ABOUT;
An aggregate denotes an array constructed from component values.

Examples:

```plaintext
type TABLE is array (1..10) of INTEGER;
A : TABLE := (7,9,5,1,3,2,4,8,6,0);

A(1) = 7 expressions which define
A(2) = 9 the values to be
A(3) = 5 associated with
... components given by
A(10) = 0 position (index
order for array components)
```
B : TABLE := (5, 4, 8, 1, others => 20);

positioned

B(1) = 5
B(2) = 4
B(3) = 8
B(4) = 1
B(5) thru B(10) = 20

C : TABLE := ( 2 | 4 | 10 => 1, others => 0 );

named components

C(1) = 0
C(2) = 1
C(3) = 0
C(4) = 1
C(5) thru C(9) = 0
C(10) = 1

An aggregate must provide values for all components.

The choice "others" stands for all components not specified by previous choices.

If used, "others" must appear last.
type MATRIX is array (INTEGER range <>, INTEGER range <>)

OF FLOAT;

NULL_MATRIX : constant MATRIX

:= ( 1..10 => (1..10 => 0.0) );

An aggregate can be used to give values to components and to
provide bounds for an array object. In this case, the
choice "others" cannot be used.

An aggregate for an n-dimensional array is written as a one-
dimensional aggregate of components that are (n-1)-
dimensional array values.
function CHANGE_COURSE ( D : DIRECTION ; T : TURN )
    return DIRECTION is

    begin
        case T is
            when LEFT => return TURN_LEFT( D );
            when RIGHT => return TURN_RIGHT( D );
            when ABOUT => return TURN_ABOUT( D );
            when NONE => return D;
        end case;

    end CHANGE_COURSE;
function MANEUVER ( OLD, NEW : DIRECTION ) return TURN is

begin

    if NEW = TURN_LEFT( OLD ) then
        return LEFT;
    elsif NEW = TURN_RIGHT( OLD ) then
        return RIGHT;
    elsif NEW = TURN_ABOUT( OLD ) then
        return ABOUT;
    else
        return NONE;
    end if;

end MANEUVER;
NAMED PARAMETER ASSOCIATION

CURRENT_DIRECTION, NEXT_DIRECTION : DIRECTION;

Equivalent subprogram calls:

MANEUVER (OLD => CURRENT_DIRECTION,
NEW => NEXT_DIRECTION);

MANEUVER (NEW => NEXT_DIRECTION,
OLD => CURRENT_DIRECTION);

Form -

formal_parameter => actual parameter
ADDITIONAL EXAMPLES OF THE USE OF ENUMERATION TYPES
type MONTH_NAME is
    ( JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
      AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER );

MONTH : MONTH_NAME ;

if MONTH = DECEMBER and Day = 31 then

    MONTH := JANUARY ;

    DAY := 1 ;

    YEAR := YEAR + 1 ;

end if ;
type MONTH_NAME is (...);

NUMBER_OF_DAYS : constant array ( MONTH_NAME ) of INTEGER := ( APRIL | JUNE | SEPTEMBER |
   NOVEMBER => 30,
   FEBRUARY => 28,
   others => 31 );

if DAY = NUMBER_OF_DAYS ( MONTH ) then
   DAY := 1 ;
   if MONTH = DECEMBER then
      MONTH := JANUARY ;
      YEAR := YEAR + 1 ;
   else
      MONTH := MONTH_NAME'SUCC ( MONTH );
   end if ;
else
   DAY := DAY + 1 ;
end if ;
-- use of an enumeration as a state indicator

function FIND_CHAR ( S : STRING; C : CHAR )
return NATURAL is

-- function to find the position of the first
-- occurrence of a character C in a string S;
-- returns S'LENGTH + 1 if C is not present;
-- ASSUMES S IS NOT NULL!

STATE : ( SEARCHING, FOUND, NOTPRESENT );
POS : NATURAL range 1..S'LENGTH;

begin
STATE := SEARCHING;
POS := 1; -- assumes S is not null
loop
  if S(POS) = C then
    STATE := FOUND;
  elsif POS = S'LENGTH then
    STATE := NOTPRESENT;
  else
    POS := POS + 1;
  end if;
  exit when STATE /= SEARCHING;
end loop;
if STATE = FOUND then
  return POS;
else -- STATE = NOTPRESENT
  return S'LENGTH + 1;
end if;
end FIND_CHAR;
begin

STATE := SEARCHING ;

loop

if ... then

... 

end if;

exit when STATE /= SEARCHING ;

end loop ;
within the loop -

if \( S(\ pos) = C \) then

\[ \text{STATE} := \text{FOUND} ; \]

elsif \( \pos = \text{S'LENGTH} \) then

\[ \text{STATE} := \text{NOTPRESENT} ; \]

else

\[ \pos := \pos + 1 ; \]

end if ;
upon exit from loop -

if STATE = FOUND then

    return POS;

else -- STATE = NOTPRESENT

    return S'LENGTH + 1;

end if;
-- This function compares two strings, which may not be of equal
-- length. Two strings are equal if they match through the length
-- of the shorter string and the longer string is blank filled
-- beyond that point.

function STRING_EQUAL (S1, S2 : STRING) return BOOLEAN is
    type SEARCH_STATE is
        (EQUAL, NOT_EQUAL, S1_LONGER, S2_LONGER, CHECKING);
    STATE : SEARCH_STATE := CHECKING;
    INDEX : INTEGER range 1..MAX(S1'LENGTH,S2'LENGTH) := 1;
EQUAL STRINGS

STRING_EQUAL ( "BEST", "BEST" ) -- TRUE

STRING_EQUAL ( "BEST", "BEAT" ) -- FALSE

STRING_EQUAL ( "BET", "BETTER" ) -- FALSE

STRING_EQUAL ( "BET", "BET" ) -- TRUE

STRING_EQUAL ( "", "" ) -- TRUE
function BLANKS (S : STRING) return BOOLEAN is
   -- Returns true only for a string of all blanks
begin
   for I in 1.. S'LENGTH loop
      if S(I) /= ' ' then
         return FALSE;
      end if;
   end loop;
   return TRUE;
end BLANKS;
begin
    -- first check for null strings
    if S1'LENGTH = 0 then
        if S2'LENGTH = 0 then
            STATE := EQUAL;
        else
            STATE := S2_LONGER;
        end if;
    elsif S2'LENGTH = 0 then
        STATE := S1_LONGER;
    end if;

    -- check the strings character by character
    while STATE = CHECKING loop
        if S1(INDEX) /= S2(INDEX) then
            STATE := NOT_EQUAL;
        elsif INDEX = S1'LENGTH then
            if INDEX = S2'LENGTH then
                STATE := EQUAL;
            else
                STATE := S2_LONGER;
            end if;
        elsif INDEX = S2'LENGTH then
            STATE := S1_LONGER;
        end if;
        INDEX := INDEX + 1;
    end loop;

    -- return with value based on current state
    case STATE is
        when EQUAL => return TRUE;
        when NOT_EQUAL => return FALSE;
        when S1_LONGER => return BLANKS(S1(INDEX..S1'LENGTH));
        when S2_LONGER => return BLANKS(S2(INDEX..S2'LENGTH));
        when CHECKING => null; -- this branch is unreachable
    end case;
end STRING_EQUAL;
This function compares two strings, which may not be of equal length. Two strings are equal if they match through the length of the shorter string and the longer string is blank filled beyond that point.

```plaintext
function STRING_EQUAL (S1, S2 : STRING) return BOOLEAN is
    type SEARCH_STATE is
        (EQUAL, NOT_EQUAL, S1_LONGER, S2_LONGER, CHECKING);
    STATE : SEARCH_STATE := CHECKING;
    INDEX : INTEGER range 1..MAX(S1'LENGTH, S2'LENGTH) := 1;

    function BLANKS (S : STRING) return BOOLEAN is
        -- Returns true only for a string of all blanks
        begin
            for I in 1..S'LENGTH loop
                if S(I) /= ' ' then
                    return FALSE;
                end if;
            end loop;
            return TRUE;
        end BLANKS;

    begin
        -- first check for null strings
        if S1'LENGTH = 0 then
            if S2'LENGTH = 0 then
                STATE := EQUAL;
            else
                STATE := S2_LONGER;
            end if;
        elsif S2'LENGTH = 0 then
            STATE := S1_LONGER;
        end if;

        -- check the strings character by character
        while STATE = CHECKING loop
            if S1(INDEX) /= S2(INDEX) then
                STATE := NOT_EQUAL;
            elsif INDEX = S1'LENGTH then
                if INDEX = S2'LENGTH then
                    STATE := EQUAL;
                else
                    STATE := S2_LONGER;
                end if;
            elsif INDEX = S2'LENGTH then
                STATE := S1_LONGER;
            end if;
            INDEX := INDEX + 1;
        end loop;
```
-- return with value based on current state
case STATE is
  when EQUAL => return TRUE;
  when NOT EQUAL => return FALSE;
  when S1 LONGER => return BLANKS(S1(INDEX..S1'LENGTH));
  when S2 LONGER => return BLANKS(S2(INDEX..S2'LENGTH));
  when CHECKING => null; -- this branch is unreachable
end case;
end STRING_EQUAL;
SUMMARY

Enumeration Types

Array Aggregates

Named Parameter Association
EXAMPLE V

OVERLOADING

and

EXCEPTIONS
OBJECTIVES

Overloading

Exceptions

Packages and Exceptions
package MATRIX_OPS is
    type MATRIX is array (INTEGER range <>, INTEGER range <>)
               of FLOAT;

    function "+" ( A : FLOAT; M : MATRIX ) return MATRIX;
    function "+" ( M1, M2 : MATRIX ) return MATRIX;
    function "*" ( A : FLOAT; M : MATRIX ) return MATRIX;
    function "*" ( M1, M2 : MATRIX ) return MATRIX;
end MATRIX_OPS;

package body MATRIX_OPS is
    function "+" ( A : FLOAT; M : MATRIX ) return MATRIX is
        TEMP : MATRIX( M'FIRST(1) .. M'LAST(1), M'FIRST(2) .. M'LAST(2) );
        begin
          for I in M'FIRST .. M'LAST loop
            for J in M'FIRST(2) .. M'LAST(2) loop
              TEMP(I,J) := A + M(I,J);
            end loop;
          end loop;
        return TEMP;
    end "+";
function "+" ( M1, M2 : MATRIX ) return MATRIX is

    TEMP : MATRIX( M1'FIRST..M1'LAST, M1'FIRST(2)..M1'LAST(2) );
    IOFFSET, JOFFSET : INTEGER;

begin

    IOFFSET := M2'FIRST(1) - M1'FIRST(1);
    JOFFSET := M2'FIRST(2) - M1'FIRST(2);

    for I in M1'FIRST(1) .. M1'LAST(1) loop
        for J in M1'FIRST(2) .. M1'LAST(2) loop
            TEMP(I,J) := M1(I,J) + M2(I + IOFFSET, J + JOFFSET);
        end loop;
    end loop;

    return TEMP;
end "+";

function "**" ( A : FLOAT; M : MATRIX ) return MATRIX is

    TEMP : MATRIX( M'FIRST(1) .. M'LAST(1), M'FIRST(2) .. M'LAST(2) );

begin

    for I in M'FIRST(1) .. M'LAST(1) loop
        for J in M'FIRST(2) .. M'LAST(2) loop
            TEMP(I,J) := A * M(I,J);
        end loop;
    end loop;

    return TEMP;
end "**";
function "**" ( M1, M2 : MATRIX ) return MATRIX is

    TEMP : MATRIX(M1'FIRST(1)..M1'LAST(1), M2'FIRST(2)..M2'LAST(2) );
    OFFSET : constant INTEGER := M2'FIRST(1) - M1'FIRST(2);

begin

    for I in M1'FIRST(1) .. M1'LAST(1) loop
        for J in M2'FIRST(2) .. M2'LAST(2) loop
            TEMP(I,J) := 0.0;
            for K in M1'FIRST(2) .. M1'LAST(2) loop
                TEMP(I,J) := TEMP(I,J) + M1(I,K) * M2(K + OFFSET, J);
            end loop;
        end loop;
    end loop;

    return TEMP;

end "**";

end MATRIX_OPS;
package MATRIX_OPS is

    type MATRIX is array ( INTEGER range <>, INTEGER range <> ) of FLOAT;

    function "+" ( A : FLOAT; M : MATRIX ) return MATRIX;
    function "+" ( M1, M2 : MATRIX ) return MATRIX;
    function "*" ( A : FLOAT; M : MATRIX ) return MATRIX;
    function "*" ( M1, M2 : MATRIX ) return MATRIX;

end MATRIX_OPS;
OVERLOADING OF OPERATIONS

package MATRIX_OPS is

  ... 
  
  function "+" ( A : FLOAT, M : MATRIX )
  return MATRIX;

  function "+" ( M1, M2 : MATRIX )
  return MATRIX;

  ...

end MATRIX_OPS;

A function named by a character string is used to define additional meaning for an operator
+ defined for any numeric type
  ( integer and real )

new meaning :

  scalar + matrix

  matrix + matrix

  character string must denote one of operators in language

  + and - permitted for unary and binary operators

  * and / permitted for binary operators

  < , > , <= , >= can be overloaded; result must be type boolean
-- use of MATRIX_OPS

declare

use MATRIX_OPS;
A, B : MATRIX( 1..10, 1..20);
C : MATRIX(11..30, 1..30);
D, E : MATRIX( 1..10, 1..30);
X, Y : FLOAT;

begin

-- assume initialization done here
A := X + B ;  -- first "+
A := 3.5 + B ;  -- first "+
A := A + B ;  -- second "+
C := Y * C ;  -- first "*
D := -9.7 * E ;  -- first "*
E := A * C ;  -- second "*
E := D + (A + B) * (5.25 * C ) ;
A := A + 1.0 ;  -- error : there is no such
                 -- "+" operation

end;  -- of example of usage
function "+" ( A : FLOAT; M : MATRIX ) return MATRIX is
  TEMP : MATRIX( M'FIRST(1)..M'LAST(1) , M'FIRST(2)..M'LAST(2) );
begin
  for I in M'FIRST .. M'LAST loop
    for J in M'FIRST(2) .. M'LAST(2) loop
      TEMP(I,J) := A + M(I,J);
    end loop;
  end loop;
  return TEMP;
end "+";
function "+" (A : FLOAT; M : MATRIX) return MATRIX is
  subtype ROWS is INTEGER range M'FIRST(1) .. M'LAST(1);
  subtype COLS is INTEGER range M'FIRST(2) .. M'LAST(2);
  TEMP : MATRIX(ROWS, COLS);
begin
  for I in ROWS loop
    for J in COLS loop
      TEMP(I,J) := A + M(I,J);
    end loop;
  end loop;
  return TEMP;
end "+";
function "+" ( A : FLOAT; M : MATRIX ) return MATRIX is

    TEMP : MATRIX ( M'FIRST(1) .. M'LAST(1),
                    M'FIRST(2) .. M'LAST(2) );

begin
    . . .
end "+";

will return TEMP; attributes taken from actual parameters

M'FIRST(i)     lower bound of i-th index
M'LAST(i)      upper bound of i-th index
Object declaration

A : MATRIX (-5..5, 1..20)

A'FIRST(1)--'
A'LAST(1)------
A'FIRST(2)--------
A'LAST(2)----------

When the declaration "TEMP : ..." is elaborated, an object having 11 rows and 20 columns will be created.
A := A + 1.0; -- SYNTAX ERROR

+ not defined for matrix
as first parameter and
scalar as second parameter

could add

function "+" ( M:MATRX; A:FLOAT )
return MATRIX is
begin
  return A + M;
end "+";

to MATRIX_OPS
function "+" ( M1, M2 : MATRIX ) return MATRIX is

    TEMP : MATRIX( M1'FIRST..M1'LAST, M1'FIRST(2)..<M1'LAST(2) );
    IOFFSET, JOFFSET : INTEGER;
begin

    IOFFSET := M2'FIRST(1) - M1'FIRST(1);
    JOFFSET := M2'FIRST(2) - M1'FIRST(2);

    for I in M1'FIRST(1) .. M1'LAST(1) loop
        for J in M1'FIRST(2) .. M1'LAST(2) loop
            TEMP(I,J) := M1(I,J) + M2(I + IOFFSET, J + JOFFSET);
            end loop;
        end loop;

    return TEMP;
end "+";
function "+" (M1,M2: MATRIX) return MATRIX is

TEMP : MATRIX ( M1'FIRST..M1'LAST,
               M1'FIRST(2)..M1'LAST(2) );

indices of returned matrix
taken from left operand

Object declarations -

    S,T : MATRIX (1..4,1..6);
    U : MATRIX (-3..0,10..15);

S + T and S + U return a
4x6 matrix with indices
    1..4 x 1..6

U + S returns a 4x6 matrix
with indices -3..0 x 10..15
discrete range for loops taken from first operand

\[
S + U \quad \text{for } I \text{ in } 1..4 \text{ loop}
\]
\[
\quad \text{for } J \text{ in } 1..6 \text{ loop}
\]
\[
\quad \ldots
\]
\[
U + S \quad \text{for } I \text{ in } -3..0 \text{ loop}
\]
\[
\quad \text{for } J \text{ in } 10..15 \text{ loop}
\]
\[
\quad \ldots
\]
Consider \( U + S \)

\[
\begin{align*}
\text{OFFSET} + \text{JOFFSET} & \\
U_{-3..0,10..15} + S_{1..4,1..6} & \\
\text{OFFSET} + \text{IOFFSET} & \\
\text{IOFFSET} := M_2^{\text{FIRST}(1)} - M_1^{\text{FIRST}(1)} & \\
& = 1 - (-3) \\
& = 4 \\
\text{JOFFSET} := M_2^{\text{FIRST}(2)} - M_1^{\text{FIRST}(2)} & \\
& = 1 - 10 \\
& = -9
\end{align*}
\]
function "*" ( A : FLOAT; M : MATRIX ) return MATRIX is

  TEMP : MATRIX( M'FIRST(1) .. M'LAST(1), M'FIRST(2) .. M'LAST(2) );

begin

  for I in M'FIRST(1) .. M'LAST(1) loop
    for J in M'FIRST(2) .. M'LAST(2) loop
      TEMP(I,J) := A * M(I,J);
    end loop;
  end loop;

  return TEMP;

end "*" ;
function "**" ( M1, M2 : MATRIX ) return MATRIX is

    TEMP : MATRIX(M1'FIRST(1) .. M1'LAST(1), M2'FIRST(2) .. M2'LAST(2));
    OFFSET : constant INTEGER := M2'FIRST(1) - M1'FIRST(2);

begin

    for I in M1'FIRST(1) .. M1'LAST(1) loop
        for J in M2'FIRST(2) .. M2'LAST(2) loop
            TEMP(I,J) := 0.0;
            for K in M1'FIRST(2) .. M1'LAST(2) loop
                TEMP(I,J) := TEMP(I,J) + M1(I,K) * M2(K + OFFSET, J);
            end loop;
        end loop;
    end loop;

    return TEMP;

end "**";
MATRIX MULTIPLICATION

\[ A_{mxn} \times B_{nxp} \rightarrow C_{mxp} \]

Product of two matrices is defined only when number of columns in first matrix is equal to the number of rows in the second.

\[
\begin{align*}
  c_{ij} &= \sum_{k=1}^{N} a_{ik} \times b_{kj} \\
  &= \sum_{k=1}^{N} a_{ik} b_{kj}
\end{align*}
\]
function "**" ( M1,M2 : MATRIX ) return MATRIX is

 TEMP: MATRIX ( M1'FIRST(1)..M1'LAST(1),
                M2'FIRST(2)..M2'LAST(2) );

Object declarations -

 S : MATRIX (1..4,1..6) ;
 T : MATRIX (1..6,1..2) ;
 U : MATRIX (1..8,1..4) ;

S * T returns a 4x2 matrix
    with indices 1..4 x 1..2

U * S returns a 8x6 matrix
    with indices 1..8 x 1..6

T * S is undefined
subprogram_specification is

declarative_part

begin

sequence_of_statements

exception

exception_handler

end;
Exception handler defines action to be taken when specific exceptions are raised.

```
declare procedure
...
begin begin
...
...
end;
```

```
begin begin
...
...

exception exception
...

end;
```
Form of exception handler

    when exception choices =>
    sequence_of_statements

exception_choices:

    exception_name

others    -- must appear last

Example:

exception

    when OBJECT_ERROR =>
    PUT ("...");

    when OVERFLOW | UNDERFLOW =>
    PUT ("...");

    when others =>
    PUT ("...");
function "+" ( M1, M2 : MATRIX )
    return MATRIX is
    ...

defined only if M1 and M2 have same number of rows and same number of columns

function "*" ( M1, M2 : MATRIX )
    return MATRIX is
    ...

defined only if number of columns of M1 is equal to number of rows of M2
package MATRIX_OPS is

    type MATRIX is array (INTEGER range <>, INTEGER range <>)
        of FLOAT;

    SIZE_ERROR : exception;

function "+" ( A : FLOAT; M : MATRIX ) return MATRIX;

function "+" ( M1, M2 : MATRIX ) return MATRIX;
    -- may raise exception SIZE_ERROR if M1 and M2
    -- are not the same size

function "**" ( A : FLOAT; M : MATRIX ) return MATRIX;

function "**" ( M1, M2 : MATRIX ) return MATRIX;
    -- may raise exception SIZE_ERROR if the number
    -- of columns of M1 is not equal to the number
    -- of rows of M2

end MATRIX_OPS;

package body MATRIX_OPS is

function "+" ( A : FLOAT; M : MATRIX ) return MATRIX is

    TEMP : MATRIX( M'first(1)..M'LAST(1) , M'FIRST(2)..M'LAST(2) );

begin
    for I in M'FIRST .. M'LAST loop
        for J in M'FIRST(2) .. M'LAST(2) loop
            TEMP(I,J) := A + M(I,J);
        end loop;
    end loop;
    return TEMP;

end "+";
function "+" ( M1, M2 : MATRIX ) return MATRIX is

-- may raise exception SIZE_ERROR

TEMP : MATRIX( M1'FIRST..M1'LAST, M1'FIRST(2)..M1'LAST(2) );
IOFFSET, JOFFSET : INTEGER;

begin

if M1'LENGTH(1) /= M2'LENGTH(1) or
   M1'LENGTH(2) /= M2'LENGTH(2) then
    raise SIZE_ERROR;
end if;

IOFFSET := M2'FIRST(1) - M1'FIRST(1);
JOFFSET := M2'FIRST(2) - M1'FIRST(2);

for I in M1'FIRST(1) .. M1'LAST(1) loop
   for J in M1'FIRST(2) .. M1'LAST(2) loop
      TEMP(I,J) := M1(I,J) + M2(I + IOFFSET, J + JOFFSET);
   end loop;
end loop;

return TEMP;

end "+";

function "**" ( A : FLOAT; M : MATRIX ) return MATRIX is

TEMP : MATRIX( M'FIRST(1) .. M'LAST(1), M'FIRST(2) .. M'LAST(2) );

begin

for I in M'FIRST(1) .. M'LAST(1) loop
   for J in M'FIRST(2) .. M'LAST(2) loop
      TEMP(I,J) := A * M(I,J);
   end loop;
end loop;

return TEMP;

end "**" ;
function "**" (M1, M2 : MATRIX) return MATRIX is

  -- may raise exception SIZE_ERROR

  TEMP : MATRIX(M1'FIRST(1) .. M1'LAST(1), M2'FIRST(2) .. M2'LAST(2));
  OFFSET : constant INTEGER := M2'FIRST(1) - M1'FIRST(2);

begin
  if M1'LENGTH(2) /= M2'LENGTH(1) then
    raise SIZE_ERROR;
  end if;

  for I in M1'FIRST(1) .. M1'LAST(1) loop
    for J in M2'FIRST(2) .. M2'LAST(2) loop
      TEMP(I, J) := 0.0;
      for K in M1'FIRST(2) .. M1'LAST(2) loop
        TEMP(I, J) := TEMP(I, J) + M1(I, K) * M2(K + OFFSET, J);
      end loop;
    end loop;
  end loop;

  return TEMP;

end "**";

end MATRIX_OPS;
package MATRIX_OPS is
    type MATRIX is array ( INTEGER range <> , INTEGER <> ) of FLOAT;
    SIZE_ERROR : exception;
    function "+" ( A : FLOAT; M : MATRIX ) return MATRIX;
    function "+" ( M1, M2 : MATRIX ) return MATRIX;
        -- may raise exception SIZE_ERROR if M1 and M2
        -- are not the same size
    function "**" ( A : FLOAT; M : MATRIX ) return MATRIX;
    function "**" ( M1, M2 : MATRIX ) return MATRIX;
        -- may raise exception SIZE_ERROR if the number
        -- of columns of M1 is not equal to the number
        -- of rows of M2
end MATRIX_OPS;
USER DEFINED EXCEPTIONS

Exception declaration

    identifier_list : exception;
    SIZE_ERROR : exception;

Raise statement

    raise exception_name;
    raise SIZE_ERROR;
package MATRIX_OPS is
    ...
    SIZE_ERROR : exception;
    ...
end MATRIX_OPS;
package body MATRIX_OPS is
    ...
    function ** ( M1, M2 : MATRIX )
        return MATRIX is
            begin
            if M1'LENGTH(2) /= M2'LENGTH(1) then
                raise SIZE_ERROR;
            end if;
            ...
            end **;
            ...
end MATRIX_OPS;
Handling Exceptions

declare
  use MATRIX_OPS;
  A,B : MATRIX (1..10,1..20);
...
begin
...
  C := A * B;   -- causes SIZE_ERROR
  E := ... ;
end;

This block does not have local handler. Should SIZE_ERROR be raised, it will be propagated to enclosing unit.
Handling Exceptions

When exception is raised and propagated to unit with local handler, execution of handler replaces execution of remainder of unit.

Handler "acts" as substitute for corresponding unit.

. handler has access to parameters
. handler can issue a return

If no handler exists for exception, program will terminate!
procedure P is
    ERROR : exception;
    ...
begin
    ...
    raise ERROR;  -- This exception is handled
      -- by El
    ...
exception
    when ERROR => ... ;  -- handler El
        ...
end P ;
Handling Exceptions

procedure P is
...
   ERROR : exception;
...
procedure Q is
begin
...
   raise ERROR;
     -- This exception is handled by E2.
...
end Q;
...
begin
...
Q;
...
exception
...
when ERROR => ...; -- handler E2
   -- After execution of the handler, Q returns
   -- normally, unless the handler executes a
   -- raise statement.
   -- Execution of "raise;" would propagate
   -- ERROR out to P, where it would be handled by E1.
end Q;
...
begin
...

P;
...
Handling Exceptions

procedure P is
  ...
  ERROR : exception;
  ...

  procedure R is
  begin
    ...
    raise ERROR;
    -- Since there are no handlers in R, its execution
    -- will be terminated and the exception will be
    -- propagated to the calling subprogram.
  end R;

  procedure Q is
  begin
    ...
    R; -- An ERROR exception raised by this call to
    -- R is handled by handler E2.
    ...
    exception
    ...
    when ERROR => ...; -- handler E2
  end Q;

begin
  ...
  Q;
  ...
  R; -- An ERROR exception raised by this call to
  -- R is handled by handler E1.
  ...
  exception
  ...
  when ERROR => ...; -- handler E1
end P;
Exceptions in Example III

procedure GET_VALID_RECORD (REC : out ITEM_RECORD; END_OF_DATA : out BOOLEAN) is

S : RECORD STRING;
LENGTH_ERROR : BOOLEAN;
begin
  loop
    GET_NEXT_RECORD (S, LENGTH_ERROR);
    if LENGTH_ERROR or else not VALID_RECORD then
      WRITE_ERROR (S);
    else
      REC := CONVERT (S);
      exit;
    end if;
  end loop;
-- exit from loop only occurs when good record found
-- or when an END_ERROR exception occurs in
-- GET_NEXT_RECORD
END_OF_DATA := FALSE;
exception
  when END_ERROR => END_OF_DATA := TRUE;
end GET_VALID_RECORD;

GET_VALID_RECORD calls GET_NEXT_RECORD
GET_NEXT_RECORD calls GET

GET is a procedure defined in the standard package TEXT_IO and END_ERROR is an exception defined in that package which can result from a call to GET.

Since there is no handler in GET_NEXT_RECORD, that procedure terminates and the exception is propagated on to GET_VALID_RECORD, where it is "handled" by the exception handler shown above.
NOTE: A normal return from GET_VALID_RECORD follows.
Exceptions in Example III

Suppose we want to terminate the loop in PROCESS_RECORDS using an exception when no more records are available. The following redefinition of RECORD_HANDLER would be appropriate.

package RECORD_HANDLER is

  type ITEM_RECORD is
    record
      ITEM_CODE : record
        PREFIX : STRING(1..2);
        NUMBER : range 0..9999;
        SUFFIX : CHARACTER;
      end;
      DESCRIPTION : STRING(1..30);
      QUANTITY : range 0..999999;
    end ITEM_RECORD

  procedure OPEN_FILES;
  procedure CLOSE_FILES;
  procedure GET_VALID_RECORD (REC : out ITEM_RECORD);

  NO_MORE_RECORDS : exception;
    -- This exception is raised by GET_VALID_RECORD
    -- when the end of the input file is encountered.

  procedure WRITE_RECORD (REC : in ITEM_RECORD);

end RECORD_HANDLER;
Exceptions in Example III

PROCESS RECORDS could depend on the exception

NO_MORE_RECORDS:

with RECORD_HANDLER;

procedure PROCESS_RECORDS is

use RECORD_HANDLER;

ITEM : ITEM_RECORD; -- defined in RECORD_HANDLER

begin

OPEN_FILES;

loop

GET_VALID_RECORD (ITEM, NO_MORE_RECORDS);

WRITE_RECORD (ITEM);

end loop;

exception

when NO_MORE_RECORDS => CLOSE_FILES;

end PROCESS_RECORDS;
Exceptions in Example III

The body of GET_Valid_RECORD changes slightly.

procedure GET_Valid_RECORD (REC : out ITEM_RECORD) is
  S : RECORD_STRING;
  LENgTH_ERROR : BOOLEAN;
begin
  loop
    GET_NEXT_RECORD (S, LENGTH_ERROR);
    if LENGTH_ERROR or else not VALID_RECORD then
      WRITE_ERROR (S);
    else
      REC := CONVERT (S);
      exit;
    end if;
  end loop;
  -- exit from loop only occurs when good record found
  -- or when an END_ERROR exception occurs in
  -- GET_NEXT_RECORD
  exception
    when END_ERROR => raise NO_MORE_RECORDS;
  end GET_VALID_RECORD;

The END_ERROR exception is handled, as before, but the handler raises the new NO_MORE_RECORDS exception defined in the specification part of this package.
SUMMARY

Overloading

Exceptions

Packages and Exceptions
EXAMPLE VI

LIST PROCESSING
OBJECTIVES

Access Types

Data Abstraction

Generics

Discriminants

Variant Records
List Processing

-- The following is an example of a list processing package, making use of access types for dynamic allocation of list nodes.
package SORTED_LIST is
  type LIST is private;
  type PRIORITY_TYPE is new NATURAL; -- derived type
  procedure CREATE (HEADER : out LIST);
  procedure INSERT (HEADER : in out LIST;
    INFO : INFO_TYPE;
    PRIORITY : PRIORITY_TYPE);
  procedure NEXT_ENTRY (HEADER : in out LIST;
    INFO : out INFO_TYPE;
    PRIORITY : out PRIORITY_TYPE);
  EMPTY_LIST : exception; -- can be raised by NEXT_ENTRY
private
  type NODE; -- incomplete type declaration
  type LIST is access NODE;
  type NODE is
    record
      PREVIOUS : LIST;
      PRIORITY : PRIORITY_TYPE;
      INFO : access INFO_TYPE;
      NEXT : LIST;
    end;
end SORTED_LIST

-- The procedures in this package maintain a list of items, sorted by priority (increasing). The procedure CREATE must be called each time a new list is desired. During the execution of a program any number of lists may exist. A call to NEXT_ENTRY returns the info and priority for the first item and removes this entry from the list.

VI.120
package body SORTED_LIST is

    procedure CREATE (HEADER : out LIST) is
    begin -- Build a dummy node to represent an empty list
        HEADER := new NODE (PRIORITY => 1, INFO => null,
                                PREVIOUS => null, NEXT => null);
        HEADER.PREVIOUS := HEADER; HEADER.NEXT := HEADER;
    end CREATE;

    procedure INSERT (HEADER : in out LIST;
                      INFO : INFO_TYPE;
                      PRIORITY : PRIORITY_TYPE) is
        PTR : LIST;
    begin
        PTR := HEADER.NEXT;
        while PTR /= HEADER and
            PRIORITY <= PTR.PRIORITY loop
            PTR := PTR.NEXT;
        end loop;
        --PTR now references the record which will follow
        --the new record in the list.
        PTR.PREVIOUS.NEXT := new NODE (PTR.PREVIOUS, PRIORITY,
                                        new INFO_TYPE(INFO), PTR);
        PTR.PREVIOUS := PTR.PREVIOUS.NEXT;
    end INSERT;

    procedure NEXT_ENTRY (HEADER : in out LIST;
                           INFO : out INFO_TYPE;
                           PRIORITY : out PRIORITY_TYPE) is
        FIRST : LIST := HEADER.NEXT;
    begin
        if FIRST = HEADER then
            raise EMPTY_LIST;
        end if;
        PRIORITY := FIRST.PRIORITY;
        INFO := FIRST.INFO.all;
        FIRST := FIRST.NEXT;
        HEADER.NEXT := FIRST;
        FIRST.PREVIOUS := FIRST;
        FIRST.PREVIOUS := HEADER;
    end NEXT_ENTRY;

    end SORTED_LIST;
INTRODUCTION TO ACCESS TYPES

LINKED LISTS

type NODE; -- incomplete type declaration;

type NODE_PTR is access NODE;

type NODE is
record
   WORD : STRING(1..3);
   NEXT : NODE_PTR;
end record;

Object declaration:

FIRST, LAST : NODE_PTR;
FIRST := new NODE ("ALL",null);

FIRST

FIRST.WORD = "ALL"
FIRST.NEXT = null

FIRST.NEXT := new NODE
( WORD => "BUT",
  NEXT => null );

FIRST

FIRST.NEXT.WORD = "BUT"
LAST := new NODE ( NEXT => null, WORD => "THE" );

FIRST

\[\text{ALL} \rightarrow \text{BUT \null} \]

LAST

\[\text{THE \null} \]

FIRST.NEXT.NEXT := LAST;

FIRST

\[\text{ALL} \rightarrow \text{BUT} \]

LAST

\[\text{THE \null} \]
To print the WORD fields of the records (assume zero or more nodes):

```
declare
   T : NODE_PTR := FIRST;
begin
   while T /= null loop
      PUT ( T.WORD );
      NEW_LINE;
      T := T.NEXT;
   end loop;
end;
```
I. DOUBLY LINKED LIST

Maintain a list of items sorted by priority (decreasing)

PROCEDURES:
CREATE
INSERT
NEXT_ENTRY
type INFO_TYPE is ...;
type PRIORITY_TYPE is ...;

type NODE;
type LIST is access NODE;

type NODE is
  record
    PREVIOUS : LIST;
    PRIORITY : PRIORITY_TYPE;
    INFO : access INFO_TYPE;
    NEXT : LIST;
  end record;

type LIST is access NODE;
PRIVATE TYPE

package SORTED_LIST is
  type LIST is private;
  procedure CREATE (...); visible
  procedure INSERT (...); part
  procedure NEXT_ENTRY (...);
  EMPTY_LIST : exception;
private
  type NODE;
  type LIST is access NODE;
  type NODE is
    record private
      ...
    end record;
end SORTED LIST;

Name of type and operations specified in visible part are available.

Names of fields are not visible.
procedure CREATE
   ( HEADER : out LIST ) is
begin
   HEADER := new LIST
         ( PRIORITY => 1,
           INFO => null,
           PREVIOUS => null,
           NEXT => null );
   HEADER.PREVIOUS := HEADER;
   HEADER.NEXT := HEADER;
end CREATE;
Procedure Insert

Before

Priority = 5

After
procedure INSERT (HEADER : in out LIST;
    INFO : INFO_TYPE;
    PRIORITY : PRIORITY_TYPE) is

    PTR : LIST;

begin

    PTR := HEADER.NEXT;

    while PTR /= HEADER and
        PRIORITY <= PTR.PRIORITY loop
        PTR := PTR.NEXT;
    end loop;

    --PTR now references the record which will follow
    --the new record in the list.

    PTR.PREVIOUS.NEXT := new NODE (PTR.PREVIOUS, PRIORITY,
        new INFO_TYPE(INFO), PTR);

    PTR.PREVIOUS := PTR.PREVIOUS.NEXT;

end INSERT;
procedure INSERT ( ...) is ...
begin
    PTR := HEADER.NEXT;
    while PTR /= HEADER and
         PRIORITY <= PTR.PRIORITY loop
      PTR := PTR.NEXT;
    end loop;

    upon exit from loop:
new LIST ( PTR.PREVIOUS,
    PRIORITY,
    new INFO_TYPE(INFO),
    PTR )
PTR.PREVIOUS.NEXT := new LIST(...);

PTR.PREVIOUS := PTR.PREVIOUS.NEXT;
INSERT at end of list

PTR /= HEADER is true
PRIORITY <= PTR.PRIORITY is true

PTR /= HEADER is false

loop terminates
PTR.PREVIOUS.NEXT := new LIST(...);

PTR.PREVIOUS := PTR.PREVIOUS.NEXT;
INSERT first item

loop terminates immediately with

PTR = HEADER

PTR.PREVIOUS.NEXT := new LIST (...);

PTR.PREVIOUS := PTR.PREVIOUS.NEXT;
PROCEDURE NEXT_ENTRY

HEADER

FIRST

PRIORITY := FIRST.PRIORITY; \( \quad = 8 \)

INFO := FIRST.INFO.all;
FIRST := FIRST.NEXT;

HEADER.NEXT := FIRST;

FIRST.PREVIOUS := HEADER;
FIRST := FIRST.NEXT;

HEADER := FIRST;

HEADER.NEXT := FIRST;

FIRST.PREVIOUS := HEADER;

FIRST
The following is an example of how SORTED_LIST might be used. The package is declared inside of this procedure so that use may be made of a local definition of INFO_TYPE.

procedure PRINT_HANDLER;

  type INFO_TYPE is
    record
      ...
    end record;

package SORTED_LIST is
  -- specification part as defined previously,
  -- using INFO_TYPE as just declared
end SORTED_LIST;

use SORTED_LIST;

PRINT_QUEUE : LIST;
PRIORITY : PRIORITY_TYPE;
DESCRIPTOR : INFO_TYPE;

package body SORTED_LIST is
  -- as defined previously
end SORTED_LIST;

begin  -- body of PRINT_HANDLER:

  CREATE (PRINT_QUEUE);

  ...

  -- assume some value has been given to DESCRIPTOR
  INSERT (PRINT_QUEUE, DESCRIPTOR, 2);

  ...

  NEXT_ENTRY (PRINT_QUEUE, DESCRIPTOR, PRIORITY);

  ...

end PRINT_HANDLER;
Example VI
Version 2
Introduction to Generics

-- A more general list processing package definition is now presented, making use of the generic definition feature.
-- Since the package does not depend on the details of INFO_TYPE, it is now supplied as a generic parameter of the package.

generic
  type INFO_TYPE is private;

package SORTED_LIST is

type LIST is private;

type PRIORITY_TYPE is new NATURAL; -- derived type

procedure CREATE (HEADER : out LIST);

procedure INSERT (HEADER : in out LIST;
  INFO : INFO_TYPE;
  PRIORITY : PRIORITY_TYPE);

procedure NEXT_ENTRY (HEADER : in out LIST;
  INFO : out INFO_TYPE;
  PRIORITY : out PRIORITY_TYPE);

EMPTY_LIST : exception; -- can be raised by NEXT_ENTRY

private

  type NODE;
  type LIST is access NODE;
  type NODE is
    record
      PREVIOUS : LIST;
      PRIORITY : PRIORITY_TYPE;
      INFO : access INFO_TYPE;
      NEXT : LIST;
    end record;

end SORTED_LIST
-- The procedures in this package maintain a list
-- of items, sorted by priority (increasing). The procedure
-- CREATE must be called each time a new list
-- is desired. During the execution of a program
-- any number of lists may exist. A call to NEXT ENTRY
-- returns the info and priority for the first item
-- and removes this entry from the list.

package body SORTED_LIST is

procedure CREATE (HEADER : out LIST) is
begin
  -- Build a dummy node to represent an empty list
  HEADER := new NODE (PRIORITY => 1, INFO => null,
                      PREVIOUS => null, NEXT => null);
  HEADER.PREVIOUS := HEADER; HEADER.NEXT := HEADER;
end CREATE;

procedure INSERT (HEADER : in out LIST; 
  INFO : INFO_TYPE;  
  PRIORITY : PRIORITY_TYPE) is
  PTR : LIST;
begin
  PTR := HEADER.NEXT;
  while PTR /= HEADER and
    PRIORITY <= PTR.PRIORITY loop
    PTR := PTR.NEXT;
  end loop;
  -- PTR now references the record which will follow
  -- the new record in the list.
  PTR.PREVIOUS.NEXT := new NODE (PTR.PREVIOUS, PRIORITY, 
                               new INFO_TYPE(INFO), PTR);
  PTR.PREVIOUS := PTR.PREVIOUS.NEXT;
end INSERT;

procedure NEXT_ENTRY (HEADER : in out LIST;  
  INFO : out INFO_TYPE;  
  PRIORITY : out PRIORITY_TYPE) is
  FIRST : LIST := HEADER.NEXT;
begin
  if FIRST = HEADER then
    raise EMPTY_LIST;
  end if;
  PRIORITY := FIRST.PRIORITY;
  INFO := FIRST.INFO.all;
  FIRST := FIRST.NEXT;
  HEADER.NEXT := FIRST;
  FIRST.PREVIOUS := HEADER;
end NEXT_ENTRY;
end SORTED_LIST;
GENERIC PROGRAM UNITS

"Models" of program units.

Can be parameterized:

Generic instantiation creates a copy (instance) of a generic program unit which can be used directly as ordinary program units.

A generic subprogram:

    generic
        type ELEMENT is private;
    procedure EXCHANGE (X,Y : in out ELEMENT) is
        TEMP : constant ELEMENT := X;
    begin
        X := Y;
        Y := TEMP;
    end SWAP;

Declarations with generic instantiation:

    procedure SWAP_INT is new EXCHANGE (INTEGER);
    procedure SWAP_CHAR is new EXCHANGE (ELEMENT => CHARACTER);

Overloading a procedure name:

    procedure SWAP is new EXCHANGE (INTEGER);
    procedure SWAP is new EXCHANGE (CHARACTER);
The package SORTED LIST may now be treated as a library package, -- with a particular type being supplied for INFO TYPE when an -- instance of the generic package is created. PRINT_HANDLER -- is now reconsidered using this new approach.

with SORTED_LIST;

procedure PRINT_HANDLER is

  type PRINT_DESCRIPTOR is
    record
      ...
      end;

  package PRINT_LIST is
    new SORTED_LIST (INFO_TYPE => PRINT_DESCRIPTOR);

  use PRINT_LIST;

  PRINT_QUEUE : LIST;
  PRIORITY : PRIORITY_TYPE;
  DESCRIPTOR : PRINT_DESCRIPTOR;

begin  -- body of PRINT_HANDLER:

  CREATE (PRINT_QUEUE);
  ...

  -- assume some value has been given to DESCRIPTOR
  INSERT (PRINT_QUEUE, DESCRIPTOR, 2);
  ...

  NEXT_ENTRY (PRINT_QUEUE, DESCRIPTOR, PRIORITY);
  ...

end PRINT_HANDLER;
Definition of generic package:

generic
  type INFO_TYPE is private;
package SORTED_LIST is
  ...
end SORTED_LIST

Instantiation of generic package:

with SORTED_LIST;
procedure PRINT_DESCRIPTOR is
  type PRINT_DESCRIPTOR is
    record
      ...
    end record;
package PRINT_LIST is
  new SORTED_LIST (INFO_TYPE => PRINT_DESCRIPTOR);
  ...
end PRINT_DESCRIPTOR;

an object of type PRINT_DESCRIPTOR
The instantiation "brings into existance" the procedures

\[
\text{PRINT\_LIST\_CREATE ( ... )};
\]
\[
\text{PRINT\_LIST\_INSERT ( ... )};
\]
and
\[
\text{PRINT\_LIST\_NEXT\_ENTRY ( ... )};
\]

which perform operations on a doubly linked list in which one component of each node is a pointer (access type) to a record to type PRINT\_DESCRIPTOR.

-- Instantiation

package L is
new SORTED\_LIST (T)

-- Procedure call

L\_INSERT( ... )

will insert a record into the list in which one component is a pointer to an object of type T
OTHER GENERIC PARAMETER FORMS

`type` identifier is `<>'; -- denotes any discrete type

generic
  `type` T is `<>';
function NEXT_IN_CYCLE (X : T) return T is
begin
  if X = T'LAST then
    return T'FIRST
  else
    return T'SUCC(X)
  end if;
end NEXT_IN_CYCLE;

type DIRECTION is (NORTH, EAST, SOUTH, WEST);
type WEEKDAY is (MON, TUES, WED, THUR, FRI);

function TURN_RIGHT is new NEXT_IN_CYCLE (DIRECTION);
function NEXT_WEEKDAY is new NEXT_IN_CYCLE (WEEKDAY);

TURN_RIGHT( EAST ) = SOUTH
TURN_RIGHT( WEST ) = NORTH

NEXT_WEEKDAY( TUES ) = WED
NEXT_WEEKDAY( FRI ) = MON
DISCRIMINANTS

Provides a form of "dynamic" parameterization; value of discriminant need not be known at translation time.

Object of record type with discriminant may be a constrained object or an unconstrained object (dynamic allocation).

Discriminant may be used
(a) as a bound of an index constraint
(b) to specify a discriminant value in a discriminant specification
(c) as a discriminant name of a variant part

Discriminant must be a discrete type
DISCRIMINANTS

Example:

```
MAX_MESSAGE_SIZE : NATURAL := 1000;

type BUFFER_TYPE ( SIZE : INTEGER range 0..MAX_MESSAGE_SIZE) is
record
   ADDRESS : ..;
   MESSAGE : STRING ( 1..SIZE);
end record;
```

**Constrained Object**

```
IN_BUFF : BUFFER_TYPE(500);

<table>
<thead>
<tr>
<th>500</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IN_BUFF.SIZE</td>
<td>IN_BUFF.ADDRESS</td>
<td>IN_BUFF.MESSAGE(1..500)</td>
</tr>
</tbody>
</table>
```

```
OUT_BUFF : BUFFER_TYPE( SIZE => 25 );

<table>
<thead>
<tr>
<th>25</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT_BUFF.SIZE</td>
<td>OUT_BUFF.ADDRESS</td>
<td>OUT_BUFF.MESSAGE(1..25)</td>
</tr>
</tbody>
</table>
```
Unconstrained Object

declare

. . . .
A_BUFFER: BUFFER_TYPE;       -- discriminant omitted
DESTINATION: . . . ;
FULL_LINE: STRING (1..MAX_MESSAGE_SIZE);
ACTUAL_LENGTH: NATURAL := 0;
begin
GET_MESSAGE (DESTINATION, FULL_LINE, ACTUAL_LENGTH);

A_BUFFER := (ACTUAL_LENGTH, DESTINATION,
FULL_LINE(1..ACTUAL_LENGTH));
. . . .
end;

If GET_MESSAGE returns a value of 475 as the value of ACTUAL_LENGTH, the effect of the assignment statement is to create the record

<table>
<thead>
<tr>
<th>475</th>
<th>value of DESTINATION</th>
<th>value of FULL_LINE(1..475)</th>
</tr>
</thead>
</table>

VI.410
A list of records, each of which have certain objects in common. The remaining components depend on the value of some other component which is called the "discriminant".
VARIANT PART

Variant part specifies alternative record components. Each variant defines the components which exist for a specific value of the discriminant.

DISCRIMINANT:

Special component of records. Discriminant must be a discrete type.

Provides a form of "dynamic" parameterization; value of discriminant need not be known at translation time.

<table>
<thead>
<tr>
<th>RECTANGLE</th>
<th>discriminant</th>
<th>LINE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>fixed part</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>variant part</td>
<td>4.8</td>
</tr>
<tr>
<td>5.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Each value of the discriminant must be represented once and only once in the set of choices.
type COORDINATES is
  record
    X, Y : FLOAT;
  end record;

type DEGREES is new FLOAT;
-- derived type; differentiate from
-- length measurements

type SHAPE_TYPE is (SQUARE, RECTANGLE, LINE, ARC, CIRCLE);

type FIGURE (SHAPE : SHAPE_TYPE) is
  record
    COLOR      : (RED, GREEN, BLUE);
    LINE_STYLE : (SOLID_LINE, DOTTED_LINE);
    POSITION   : COORDINATES;
    ANGLE      : DEGREES;

    case SHAPE is
      when SQUARE  => SIZE     : FLOAT;
      when RECTANGLE => HEIGHT, WIDTH : FLOAT;
      when LINE    => LENGTH   : FLOAT;
      when ARC     => RADIUS   : FLOAT;
      when ARC     => ARC_LENGTH : DEGREES;

      when CIRCLE  => DIAMETER : FLOAT;
    end case;

  end record;
RECORD AGGREGATES

Using positional notation:

\[(\text{RECTANGLE, RED, SOLID\_LINE, } (1.5, 3.4), 45.0, 2.5, 5.0)\]

discriminant must appear first

Using named components

\[(\text{COLOR } \rightarrow \text{ RED, LINE\_STYLE } \rightarrow \text{ SOLID\_LINE,}
\hspace{1cm}\text{POSITION } \rightarrow (1.5, 3.4),
\hspace{1cm}\text{ANGLE } \rightarrow 45.0, \hspace{1cm}\text{SHAPE } \rightarrow \text{RECTANGLE,}
\hspace{1cm}\text{HEIGHT } \rightarrow 2.5, \hspace{1cm}\text{WIDTH } \rightarrow 5.0)\]
An Application

type ITEM;

type POINTER is access ITEM;

type ITEM is
record
  NEXT ITEM: POINTER;
  COMPONENT: FIGURE;
end record;

PICTURE: POINTER;

PICTURE := new ITEM (null, (RECTANGLE, ..., 2.5, 5.0));
PICTURE.NEXT_ITEM := new ITEM (null, (LINE, ..., 4.8));

--- null
PICTURE
RECTANGLE
...
2.5
5.0

LINE
...
4.8

PICTURE.COMPONENT.SHAPE = RECTANGLE --- reference
PICTURE.COMPONENT.HEIGHT := 3.5; --- assignment
PICTURE.COMPONENT.DIAMETER --- illegal reference
PICTURE.COMPONENT.SHAPE := CIRCLE --- illegal assignment

VI.470
SUMMARY

Access Types

Data Abstraction

Generics

Discriminants

Variant Records
EXAMPLE VII

Fundamentals of Tasking
OBJECTIVES

Task Concepts
A Fault Warning Procedure

procedure ANNOUNCE_FAULT (FAULT_CODE : INTEGER) is

  task RING_WARNING_BELL;
  task FLASH_RED_LIGHT;
  task PRINT_MESSAGE;

  task body RING_WARNING_BELL is
    ...
  end RING_WARNING_BELL;

  task body FLASH_RED_LIGHT is
    ...
  end FLASH_RED_LIGHT;

  task body PRINT_MESSAGE is
    ...
  end PRINT_MESSAGE;

begin -- body of procedure

  -- wait for tasks to do their work
  -- order of execution is unimportant

end ANNOUNCE_FAULT;
function SUM ARRAYS (A,B : FLOAT_ARRAY)
return FLOAT

SUM_OF_A = A(A-FIRST) +...+ A(A-LAST)
SUM_OF_B = B(B-FIRST) +...+ B(B-LAST)

Tasks SUM_A and SUM_B can be processed in parallel.
They are independent processes.
Each involves simple sequential processes.
No inter-process communication and no sharing of data.
function SUM ARRAYS (A, B : FLOAT ARRAY) return FLOAT is
  -- This is an example of tasks which can run in parallel
  -- because they do not interact.
  SUM_OF_A, SUM_OF_B : FLOAT := 0.0;
begin
  declare -- a block to contain the tasks
    task SUM_A; -- simplest possible task declaration
    task SUM_B; -- another, to run in parallel
  task body SUM_A is -- corresponds to a package body
    begin
      for I in A'FIRST .. A'LAST loop
        SUM_OF_A := SUM_OF_A + A(I);
      end loop;
      end SUM_A;
  task body SUM_B is
    begin
      for I in B'FIRST .. B'LAST loop
        SUM_OF_B := SUM_OF_B + B(I);
      end loop;
      end SUM_B;
    begin -- body of block
      null;
      -- This block will not terminate until both tasks terminate
      -- because they are declared in the block.
    end;
  return SUM_OF_A + SUM_OF_B;
end SUM ARRAYS;
  -- This example can be generalized to involve any number of arrays
  -- and tasks, with one task being declared for each array.
function SUM ARRAYS (A, B : FLOAT ARRAY)
  return FLOAT is
    SUM_OF_A, SUM_OF_B : FLOAT := 0.0;
begin
  declare
    ... -- task declarations
    ... -- task bodies
  begin
    -- empty body (of block)
  end
  return SUM_OF_A + SUM_OF_B;
end SUM ARRAYS;

Elaboration of the task bodies causes their initiation.
Only when tasks declared within block terminate will the
block terminate.
Task Specification

```
task [type] identifier
  is entry - declaration \n  entry - declaration
  ...
  \ optional
  entry declaration
  end identifier
```

A single task can be declared by a task specification, as has been done in this example,

or

A task type can be declared, allowing any number of variables of that type to be created.

Task types allow the inclusion of tasks in any data structure and dynamic creation of tasks using access types which reference tasks.
Example of Task Types

task type RESOURCE is
  entry SEIZE;
  entry RELEASE;
end RESOURCE;
.
.
SINGLE : RESOURCE;
POOL : array (1..10) of RESOURCE.
.
.
SINGLE.SEIZE
POOL(K).RELEASE
EXAMPLE VIII

TASK INTERACTIONS
OBJECTIVES

Entries
Accept Statements
Rendezvous
Task Attributes
Select Statements
Example VIII
Version 1
Task Interactions

-- An example of cooperating tasks running in parallel.

BLOCK_LENGTH : constant INTEGER := 100;
type BLOCK is array (1..BLOCK_LENGTH) of INTEGER;

task PRODUCE_BLOCK;
   -- A task which produces blocks of data items from any source.
   -- Each block is BLOCK_LENGTH data items long.

task CONSUME_ITEM;
   -- A task which processes data one item at a time.
   -- Structure of data blocks is unimportant to this task.

task BLOCK_TO_ITEM is
   -- A task to allow PRODUCE_BLOCK to feed CONSUME_ITEM.
   entry SEND_BLOCK (B : in BLOCK);
      -- A call to SEND_BLOCK is accepted first.
   entry GET_ITEM (ITEM : out INTEGER);
      -- 100 (BLOCK_LENGTH) calls to GET_ITEM are then accepted
      -- before looping back to the accept for SEND_BLOCK.
end BLOCK_TO_ITEM;
task body BLOCK_TO_ITEM is
   BUFFER : BLOCK;
begin
   loop -- forever
      accept SEND_BLOCK (B : in BLOCK) do
         BUFFER := B;
      end SEND_BLOCK;
      for I in 1..BLOCK_LENGTH loop
         accept GET_ITEM (ITEM : out INTEGER) do
            ITEM := BUFFER(I);
         end GET_ITEM;
         end loop;
   end loop;
end BLOCK_TO_ITEM;

-----------------------------

task body PRODUCE BLOCK is
   MY BLOCK : BLOCK;
begin
   loop
      -- fill MY BLOCK from somewhere
      BLOCK_TO_ITEM.SEND_BLOCK (MY_BLOCK);
   end loop;
end PRODUCE BLOCK;

-----------------------------

task body CONSUME ITEM is
   NEXT_ITEM : INTEGER;
begin
   loop
      BLOCK_TO_ITEM.GET_ITEM (NEXT_ITEM);
      -- consume NEXT_ITEM
   end loop;
end CONSUME ITEM;
task BLOCK_TO_ITEM is
  -- task specification
  -- contains entry declarations only
end BLOCK_TO_ITEM;

task body BLOCK_TO_ITEM is
  -- declarative part
begin
  -- sequence of statements
end BLOCK_TO_ITEM;
task body PRODUCE_BLOCK is

... -- fill MY BLOCK from somewhere
...

BLOCK_TO_ITEM.SEND_BLOCK(MY_BLOCK); -- entry call
...
end PRODUCE_BLOCK;

end BLOCK_TO_ITEM;

...
fill MY_BLOCK from somewhere

BLOCK TO ITEM.
SEND BLOCK (MY BLOCK)

accept SEND BLOCK (B : in BLOCK)

RENEZVOUS
BUFFER := B
executed
ENTRY DECLARATION
and
ENTRY CALL

ENTRY declaration
Similar to a procedure declaration in syntax
Can be declared only in a task specification

ENTRY call
Same syntax as subprogram calls
**ACCEPT STATEMENT**

```
accept entry_name

formal_part (optional)

do sequence_of_statements end (optional)
```

**formal_part**

analogous to subprogram formal_part; specifies parameters, their modes and types

**do sequence_of_statements end**

when rendezvous occurs (entry has been called and accept statement is reached) sequence_of_statements is executed
task body BLOCK_TO_ITEM is
    BUFFER : BLOCK;
begin
    loop -- forever
        accept SEND BLOCK (B : in BLOCK) do
            BUFFER := B;
        end SEND BLOCK;
        for I in 1..BLOCK_LENGTH loop
            accept GET ITEM (ITEM : out INTEGER) do
                ITEM := BUFFER(I);
            end GET ITEM;
        end loop;
    end loop;
end BLOCK_TO_ITEM;

-----------------------------

task body PRODUCE_BLOCK is
    MY BLOCK : BLOCK;
begin
    loop
        -- fill MY BLOCK from somewhere
        --
        BLOCK_TO_ITEM SEND BLOCK (MY BLOCK);
    end loop;
end PRODUCE_BLOCK;

-----------------------------

task body CONSUME_ITEM is
    NEXT ITEM : INTEGER;
begin
    loop
        BLOCK_TO_ITEM GET ITEM (NEXT ITEM);
        -- consume NEXT ITEM
        --
        NEXT ITEM;
    end loop;
end CONSUME_ITEM;
task body BLOCK_TO_ITEM is
  BUFFER : BLOCK;
begin
  loop -- forever
    accept SEND_BLOCK (B : in BLOCK) do
      BUFFER := B;
    end SEND_BLOCK;
    for I in 1..BLOCK LENGTH loop
      accept GET_ITEM (ITEM : out INTEGER) do
        ITEM := BUFFER(I);
      end GET_ITEM;
    end loop;
  end loop;
end BLOCK_TO_ITEM;

---------------------------------

task body PRODUCE_BLOCK is
  MY BLOCK : BLOCK;
begin
  loop
    -- fill MY BLOCK from somewhere
    BLOCK_TO_ITEM.SEND_BLOCK (MY BLOCK);
  end loop;
end PRODUCE_BLOCK;

---------------------------------

task body CONSUME_ITEM is
  NEXT_ITEM : INTEGER;
begin
  loop
    BLOCK_TO_ITEM.GET_ITEM (NEXT_ITEM);
    -- consume NEXT_ITEM
  end loop;
end CONSUME_ITEM;
task body BLOCK_TO_ITEM is
    BUFFER : BLOCK;
begin
    loop -- forever
        accept SEND_BLOCK (B : in BLOCK) do
            BUFFER := B;
        end SEND_BLOCK;
        for I in 1..BLOCK_LENGTH loop
            accept GET_ITEM (ITEM : out INTEGER) do
                ITEM := BUFFER(I);
            end GET_ITEM;
        end loop;
    end loop;
end BLOCK_TO_ITEM;

-----------------------------

task body PRODUCE_BLOCK is
    MY BLOCK : BLOCK;
begin
    loop
        -- fill MY_BLOCK from somewhere
        BLOCK_TO_ITEM.SEND_BLOCK (MY_BLOCK);
    end loop;
end PRODUCE_BLOCK;

-----------------------------

task body CONSUME_ITEM is
    NEXT_ITEM : INTEGER;
begin
    loop
        BLOCK_TO_ITEM.GET_ITEM (NEXT_ITEM);
        -- consume NEXT_ITEM
    end loop;
end CONSUME_ITEM;
task body BLOCK TO ITEM is
    BUFFER : BLOCK;
begin
    loop -- forever
        accept SEND BLOCK (B : in BLOCK) do
            BUFFER := B;
            end SEND BLOCK;
        for I in 1..BLOCK LENGTH loop
            accept GET ITEM (ITEM : out INTEGER) do
                ITEM := BUFFER(I);
                end GET ITEM;
        end loop;
    end loop;
end BLOCK TO ITEM;
-----------------------------

task body PRODUCE BLOCK is
    MY BLOCK : BLOCK;
begin
    loop
        -- fill MY BLOCK from somewhere
        BLOCK TO ITEM SEND BLOCK (MY BLOCK);
    end loop;
end PRODUCE BLOCK;
-----------------------------

task body CONSUME ITEM is
    NEXT ITEM : INTEGER;
begin
    loop
        BLOCK TO ITEM GET ITEM (NEXT ITEM);
        -- consume NEXT ITEM
    end loop;
end CONSUME ITEM;
task body BLOCK_TO_ITEM is
  BUFFER : BLOCK;
begin
  loop -- forever
    accept SEND_BLOCK (B : in BLOCK) do
      BUFFER := B;
      end SEND_BLOCK;
    for I in 1..BLOCK_LENGTH loop
      accept GET_ITEM (ITEM : out INTEGER) do
        ITEM := BUFFER(I);
      end GET_ITEM;
    end loop;
  end loop;
end BLOCK_TO_ITEM;

-----------------------------

task body PRODUCE_BLOCK is
  MY_BLOCK : BLOCK;
begin
  loop
    -- fill MY_BLOCK from somewhere
    BLOCK_TO_ITEM.SEND_BLOCK (MY_BLOCK); <<========
  end loop;
end PRODUCE_BLOCK;

-----------------------------

task body CONSUME_ITEM is
  NEXT_ITEM : INTEGER;
begin
  loop
    BLOCK_TO_ITEM.GET_ITEM (NEXT_ITEM); <<========
    -- consume NEXT_ITEM
  end loop;
end CONSUME_ITEM;

-----------------------------
task body BLOCK_TO_ITEM is
  BUFFER : BLOCK;
begin
  loop -- forever
    accept SEND_BLOCK (B : in BLOCK) do
      BUFFER := B;
      end SEND_BLOCK;
    for I in 1..BLOCK_LENGTH loop
      accept GET_ITEM (ITEM : out INTEGER) do
        ITEM := BUFFER(I);
      end GET_ITEM;
    end loop;
  end loop;
end BLOCK_TO_ITEM;

------------------------------------------

task body PRODUCE_BLOCK is
  MY_BLOCK : BLOCK;
begin
  loop
    -- fill MY BLOCK from somewhere
    
    BLOCK_TO_ITEM.SEND_BLOCK (MY_BLOCK);  
    end loop;
end PRODUCE_BLOCK;

------------------------------------------

task body CONSUME_ITEM is
  NEXT_ITEM : INTEGER;
begin
  loop
    BLOCK_TO_ITEM.GET_ITEM (NEXT_ITEM);
    -- consume NEXT_ITEM
    
    end loop;
end CONSUME_ITEM;
**VERSION 2 - STRUCTURE**

-- An example of cooperating tasks running in parallel,  
-- within a complete program

procedure MAIN;
  
  task BLOCK TO ITEM is ... ;  
  task PRODUCE BLOCK;  
  task CONSUME ITEM;

  task body BLOCK TO ITEM is ... ;  
  task body PRODUCE BLOCK is ... ;  
  task body CONSUME ITEM is ... ;

begin -- body of MAIN
  
  loop
    delay 15.0 * SECONDS;

    exit when PRODUCE BLOCK'TERMINATED  
    and CONSUME ITEM'TERMINATED;
  end loop;

  abort BLOCK_TO_ITEM;

end MAIN;
Task Bodies

```plaintext

task body PRODUCE BLOCK is
   MY BLOCK : BLOCK;
   NO_MORE_BLOCKS : BOOLEAN := FALSE;
begin
   loop
      -- fill MY BLOCK from somewhere
      
      if NO_MORE_BLOCKS THEN
         -- Call SEND BLOCK with some indication of end
         -- of data, for example a block of negative values.
         exit;
      end if;
      BLOCK TO ITEM SEND BLOCK (MY BLOCK);
   end loop;
end PRODUCE BLOCK;

task body CONSUME ITEM is
   NEXT ITEM : INTEGER;
begin
   loop
      BLOCK TO ITEM GET ITEM (NEXT ITEM);
   exit when NEXT ITEM < 0;
   -- consume NEXT ITEM
      
   end loop;
end CONSUME ITEM;

task body BLOCK TO ITEM is
   BUFFER : BLOCK;
begin
   loop -- forever
      accept SEND BLOCK (B : in BLOCK) do
         BUFFER := B;
      end SEND BLOCK;
      for I in 1..BLOCK LENGTH loop
         accept GET ITEM (ITEM : out INTEGER) do
            ITEM := BUFFER(I);
         end GET ITEM;
      end loop;
   end loop;
end BLOCK TO ITEM;
```

VIII.260
TASK AND ENTRY ATTRIBUTES

For a task T, the following attributes are defined:

T'TERMINATED of type BOOLEAN - initially equal to FALSE
when a task is created and becomes TRUE when
the task terminates

T'STACK_SIZE indicates the number of storage units
allocated for the task (an integer number)

T'PRIORITY of predefined type PRIORITY
Defined in package STANDARD:

    subtype PRIORITY is INTEGER range implementation_defined;

PRIORITY is set by the optional appearance of
    pragma PRIORITY (static_expression);
somewhere within a task specification.

If processor resources are shared, an eligible task
with the highest priority is executed.

The priority of a task is static.

For an entry E of Task T, the following attribute can be used
within the body of task T:

E'COUNT The number of entry calls presently queued
on the queue associated with entry E.
An integer number.
The DELAY Statement

Suspends the task which executes it for at least the given time interval.

```
delay simple_expression;
```

SECONDS is a predefined constant defined in STANDARD package (implementation defined). It gives the number of basic time units in one second.
The ABORT Statement

Example:

```plaintext
abort BLOCK_TO_ITEM;
```

Causes unconditional asynchronous termination of task(s).

If a task calling an entry is abnormally terminated, it is removed from the entry queue; if the rendezvous is already in progress, the calling task is terminated but the task executing the accept statement is allowed to complete the rendezvous normally.

If there are pending entry calls for the entries of a task that is abnormally terminated, an exception TASKING_ERROR is raised for each calling task at the point where it calls the entry, including for a task presently engaged in a rendezvous, if any.

ABORT statements are almost never needed and should only be used when no other feature can do a job.
Example VIII
Version 2

-- An example of cooperating tasks running in parallel,
-- within a complete program.

procedure MAIN;

  BLOCK_LENGTH : constant INTEGER := 100;
type BLOCK is array (1..BLOCK_LENGTH) of INTEGER;

task PRODUCE_BLOCK;
  -- A task which produces blocks of data items from any source.
  -- Each block is BLOCK_LENGTH data items long.

task CONSUME_ITEM;
  -- A task which processes data one item at a time.
  -- Structure of data blocks is unimportant to this task.

task BLOCK TO ITEM is
  -- A task to allow PRODUCE BLOCK to feed CONSUME ITEM.
  entry SEND BLOCK (B : in BLOCK);
  entry GET ITEM (ITEM : out INTEGER);
end BLOCK TO ITEM;

task body BLOCK TO ITEM is
  BUFFER : BLOCK;
begin
  loop -- forever
    accept SEND BLOCK (B : in BLOCK) do
      BUFFER := B;
      end SEND BLOCK;
    for I in 1..BLOCK_LENGTH loop
      accept GET ITEM (ITEM : out INTEGER) do
        ITEM := BUFFER(I);
      end GET ITEM;
    end loop;
  end loop;
end BLOCK TO ITEM;
task body PRODUCE_BLOCK is
  MY_BLOCK : BLOCK;
  NO_MORE_BLOCKS : BOOLEAN := FALSE;
begin
  loop
    -- fill MY_BLOCK from somewhere
    .
    -- NO_MORE_BLOCKS may be changed in here
    .
    if NO_MORE_BLOCKS then
      -- Call SEND BLOCK with some indication of end
      -- of data, for example a block of negative values.
      exit;
    end if;
    BLOCK_TO_ITEM.SEND_BLOCK (MY BLOCK);
  end loop;
end PRODUCE_BLOCK;

task body CONSUME_ITEM is
  NEXT_ITEM : INTEGER;
begin
  loop
    BLOCK_TO_ITEM.GET_ITEM (NEXT_ITEM);
    exit when NEXT_ITEM < 0;
    -- consume NEXT_ITEM
    .
  end loop;
end CONSUME_ITEM;

begin -- body of main
  -- There is nothing to be done in this body, but it
  -- will not terminate until all three tasks terminate.
  -- However, BLOCK_TO_ITEM loops forever.
  -- A possible solution is to wait for the other two:
  loop
    delay 15.0 * SECONDS;
    exit when PRODUCE_BLOCK.TERMINATED
    and CONSUME_ITEM.TERMINATED;
  end loop;

  abort BLOCK_TO_ITEM;
end MAIN;
--- An example of cooperating tasks running in parallel, 
--- within a complete program with improved termination.

procedure MAIN;
    task BLOCK_TO_ITEM is ...
    task body BLOCK_TO_ITEM is ...
begin -- body of MAIN
declare
    task PRODUCE BLOCK;
    task CONSUME_ITEM;
    task body PRODUCE_BLOCK is ...
    task body CONSUME_ITEM is ...
begin -- body of block
    null;
    -- This block will terminate only after the two tasks
    -- declared within it terminate. Each explicitly does
    -- so, thus exit from this block is guaranteed and only
    -- BLOCK_TO_ITEM will still be active at that time.
end;

-- BLOCK_TO_ITEM must now be terminated to enable the
-- termination of this procedure.
abort BLOCK_TO_ITEM;
end MAIN;
Example VIII
Version 3

-- An example of cooperating tasks running in parallel,
-- within a complete program with improved termination.

procedure MAIN;

BLOCK_LENGTH : constant INTEGER := 100;

type BLOCK is array (1..BLOCK_LENGTH) of INTEGER;

task BLOCK_TO_ITEM is
  -- A task to allow PRODUCE BLOCK to feed CONSUME ITEM.
  entry SEND_BLOCK (B : in BLOCK);
  entry GET_ITEM (ITEM : out INTEGER);
end BLOCK_TO_ITEM;

task body BLOCK_TO_ITEM is
  BUFFER : BLOCK;
  begin
    loop -- forever
      accept SEND_BLOCK (B : in BLOCK) do
        BUFFER := B;
        end SEND_BLOCK;
      for I in 1..BLOCK_LENGTH loop
        accept GET_ITEM (ITEM : out INTEGER) do
          ITEM := BUFFER(I);
          end GET_ITEM;
        end loop;
      end loop;
    end BLOCK_TO_ITEM;
  begin -- body of MAIN
    declare -- a block to declare the other two tasks
    task PRODUCE_BLOCK;
      -- A task which produces blocks of data items from any
      -- source. Each block is BLOCK_LENGTH data items long.
    task CONSUME_ITEM;
      -- A task which processes data one item at a time.
      -- Structure of data blocks is unimportant to this task.
task body PRODUCE BLOCK is
  MY BLOCK : BLOCK;
  NO MORE BLOCKS : BOOLEAN := FALSE;
begin
  loop
    -- fill MY BLOCK from somewhere
    ...
    if NO MORE BLOCKS then
      -- Call SEND BLOCK with some indication of end
      -- of data, for example a block of negative values.
      exit;
    end if;
    BLOCK TO ITEM SEND BLOCK (MY BLOCK);
  end loop;
end PRODUCE BLOCK;

task body CONSUME ITEM is
  NEXT ITEM : INTEGER;
begin
  loop
    BLOCK TO ITEM GET ITEM (NEXT ITEM);
    exit when NEXT ITEM < 0;
    -- consume NEXT ITEM
    ...
  end loop;
end CONSUME ITEM;

begin -- body of block
  null;
  -- This block will terminate only after the two tasks,
  -- declared within it terminate. Each explicitly does
  -- so, thus exit from this block is guaranteed and only
  -- BLOCK TO ITEM will still be active at that time.
end;

-- BLOCK TO ITEM must now be terminated to enable the
-- termination of this procedure.

abort BLOCK TO ITEM;
end MAIN;

VIII.340
The previous example is now modified to allow BLOCK TO ITEM to buffer several blocks if PRODUCE_BLOCK gets ahead of CONSUME_ITEM.

procedure MAIN;

BLOCK_LENGTH : ... ;
type BLOCK is ... ;
task BLOCK_TO_ITEM is ... ;
task body BLOCK_TO_ITEM ...

begin -- body of MAIN

declare

task PRODUCE_BLOCK;
task CONSUME_ITEM;
task body PRODUCE_BLOCK is ... ;
task body CONSUME_ITEM is ... ;

begin -- body of block

end;

abort BLOCK_TO_ITEM;

end MAIN;
Use of a Block Buffer

BLOCK_LENGTH : constant INTEGER := 100;
type BLOCK is array (1..BLOCK_LENGTH) of INTEGER;

BUFFER_SIZE : constant INTEGER := 10;
BUFFER : array (1..BUFFER_SIZE) of BLOCK;

IN_INDEX

OUT_INDEX

BLOCK_COUNT = 4

The filling (production) of blocks and the use (consumption) of items can be carried out in parallel.
Several blocks may be buffered.
SELECT STATEMENT

Selective Wait

```
select
    alternative_1
    or alternative_2  \>
    ...
    or alternative_n  /
else
    sequence_of_statements  /
end select;
```

Each alternative is composed of

1. (optional) "guard": when condition =>
2. accept_statement
3. (optional) sequence_of_statements
Selective Wait - Open Alternatives

```
select
    accept entry_name_1;
    or accept entry_name_2;
    ...
    or accept entry_name_n;
end select;
```

- Select one of the open alternatives (accept statements) if a corresponding rendezvous is possible. An alternative is "open" if there is no guard. Rendezvous is possible when a corresponding entry call has been issued by another task.

- When several alternative rendezvous are possible and/or several open alternatives start with an accept statement for the same entry one of the alternatives will be selected at random.

- If no alternative can be immediately selected, task waits until alternative can be selected.
Selective Wait - Use of Guards

```plaintext
select
  when guard_1 =>
    accept entry_name_1;
  or when guard_2 =>
    accept entry_name_2;
  or accept entry_name_3;
  ...
end select;
```

An alternative with a guard is open if the corresponding condition is true.
Body of BLOCK_TO_ITEM

task body BLOCK_TO_ITEM is

BUFFER_SIZE : constant INTEGER := 10;
BUFFER : array (1..BUFFER_SIZE) of BLOCK;
BLOCK_COUNT : INTEGER range 0 .. BUFFER_SIZE := 0;
IN_INDEX, OUT_INDEX : INTEGER range 1 .. BUFFER_SIZE := 1;
ITEM_INDEX : INTEGER range 1 .. BLOCK_LENGTH := -1;

begin
  loop -- forever
    select
      when BLOCK_COUNT < BUFFER_SIZE =>
        accept SEND_BLOCK (B : in BLOCK) do
          BUFFER(IN_INDEX) := B;
        end SEND_BLOCK;
        IN_INDEX := IN_INDEX mod BUFFER_SIZE + 1;
        BLOCK_COUNT := BLOCK_COUNT + 1;
      or when BLOCK_COUNT > 0 =>
        accept GET_ITEM (ITEM : out INTEGER) do
          ITEM := BUFFER(OUT_INDEX, ITEM_INDEX);
        end GET_ITEM;
        ITEM_INDEX := ITEM_INDEX mod BLOCK_LENGTH + 1;
        if ITEM_INDEX = 1 then
          -- a block has been consumed
          OUT_INDEX := OUT_INDEX mod BUFFER_SIZE + 1;
          BLOCK_COUNT := BLOCK_COUNT - 1;
        end if;
    end select;
  end loop;
end BLOCK_TO_ITEM;
Example VIII
Version 4

-- The previous example is now modified to allow
-- BLOCK TO ITEM to buffer several blocks if PRODUCE_BLOCK
-- gets ahead of CONSUME_ITEM.

procedure MAIN is

    BLOCK_LENGTH : constant INTEGER := 100;
    type BLOCK is array (1..BLOCK_LENGTH) of INTEGER;

    task BLOCK TO ITEM is
        -- A task to allow PRODUCE_BLOCK to feed CONSUME_ITEM.
        entry SEND BLOCK (B : in BLOCK);
        entry GET ITEM (ITEM : out INTEGER);
    end BLOCK_TO_ITEM;

    task body BLOCK TO ITEM is
        BUFFER SIZE : constant INTEGER := 10;
        BUFFER : array (1..BUFFER_SIZE) of BLOCK;
        BLOCK COUNT : INTEGER range 0 .. BUFFER_SIZE := 0;
        IN_INDEX, OUT_INDEX : INTEGER range 1 .. BUFFER_SIZE := 1;
        ITEM_INDEX : INTEGER range 1 .. BLOCK_LENGTH := 1;

        begin
            loop -- forever
                select
                    when BLOCK_COUNT < BUFFER_SIZE =>
                        accept SEND BLOCK (B : in BLOCK) do
                            BUFFER(IN_INDEX) := B;
                        end SEND_BLOCK;
                        IN_INDEX := IN_INDEX mod BUFFER_SIZE + 1;
                        BLOCK_COUNT := BLOCK_COUNT + 1;
                    or when BLOCK_COUNT > 0 =>
                        accept GET ITEM (ITEM : out INTEGER) do
                            ITEM := BUFFER(OUT_INDEX, ITEM_INDEX);
                        end GET_ITEM;
                        ITEM_INDEX := ITEM_INDEX mod BLOCK_LENGTH + 1;
                        if ITEM_INDEX = 1 then
                            -- a block has been consumed
                            OUT_INDEX := OUT_INDEX mod BUFFER_SIZE + 1;
                            BLOCK_COUNT := BLOCK_COUNT - 1;
                        end if;
                end select;
            end loop;
        end BLOCK_TO_ITEM;
begin -- body of MAIN

declare -- a block to declare the other two tasks

task PRODUCE BLOCK;
  -- A task which produces blocks of data items from any
  -- source. Each block is BLOCK_LENGTH data items long.

task CONSUME ITEM;
  -- A task which processes data one item at a time.
  -- Structure of data blocks is unimportant to this task.

task body PRODUCE BLOCK is
  MY BLOCK : BLOCK;
  NO MORE BLOCKS : BOOLEAN := FALSE;
begin
  loop
    -- fill MY BLOCK from somewhere
    ...
    if NO MORE BLOCKS then
      -- Call SEND BLOCK with some indication of end
      -- of data, for example a block of negative values.
      exit;
    end if;
    BLOCK TO ITEM.SEND_BLOCK (MY BLOCK);
  end loop;
end PRODUCE_BLOCK;

task body CONSUME ITEM is
  NEXT ITEM : INTEGER;
begin
  loop
    BLOCK TO ITEM.GET ITEM (NEXT ITEM);
    exit when NEXT ITEM < 0;
    -- consume NEXT ITEM
  end loop;
end CONSUME_ITEM;

begin -- body of block
  null;
  -- This block will terminate only after the two tasks
  -- declared within it terminate. Each explicitly does
  -- so, thus exit from this block is guaranteed and only
  -- BLOCK TO ITEM will still be active at that time.
end;
-- BLOCK TO ITEM must now be terminated to enable the
-- termination of this procedure.
abort BLOCK TO ITEM;
end MAIN;
Selective Wait - Else Part

```plaintext
select
  alternative_1;
or alternative_2;
...
or alternative_n;
else
  sequence_of_statements
end select;
```

- Alternative selected as before.
- If no alternative can be immediately selected, the else part is executed.
Selective Wait - SELECT ERROR

```plaintext
select
  guard_1 =>
    accept entry_name_1;
  or guard_2 =>
    accept entry_name_2;
  or guard_3 =>
    accept entry_name_3;
end select;
```

If all alternatives are closed (all guards are FALSE) then the exception SELECT_ERROR is raised.
Forms of Alternatives

```
when condition =>
  accept entry_name
  do sequence_of_statements end
  sequence_of_statements

when condition =>
  delay_statement
  sequence_of_statements

when condition =>
  terminate
```

An open alternative starting with a delay statement will be selected if no other alternative has been selected before the specified time interval has elapsed.

A selective wait can contain at most one terminate alternative. An open terminate alternative will be selected only if the end of the program unit containing the task has been reached and all other tasks depending on that program unit have either terminated or are waiting at a selective wait with a terminate alternative.

An alternative starting with a delay statement, a terminate alternative and an else part are mutually exclusive.
select
  when guard_1 =>
    entry_name_1;
  when guard_2 =>
    entry_name_2;
or
  when guard-3 =>
    delay expression-1
  or
    delay expression-2;
end select;

Both could be open
only the one
with the
shortest time
interval is
selected.
task body BLOCK_TO_ITEM is
  BUFFER_SIZE : constant INTEGER := 10;
  BUFFER : array (1..BUFFER_SIZE) of BLOCK;
  BLOCK_COUNT : INTEGER range 0 .. BUFFER_SIZE := 0;
  IN_INDEX, OUT_INDEX : INTEGER range 1 .. BUFFER_SIZE := 1;
  ITEM_INDEX : INTEGER range 1 .. BLOCK_LENGTH := 1;
begin
  loop -- forever
    select
      when BLOCK_COUNT < BUFFER_SIZE =>
        accept SEND_BLOCK (B : in BLOCK) do
          BUFFER(IN_INDEX) := B;
          end SEND_BLOCK;
          IN_INDEX := IN_INDEX mod BUFFER_SIZE + 1;
          BLOCK_COUNT := BLOCK_COUNT + 1;
      or when BLOCK_COUNT > 0 =>
        accept GET_ITEM (ITEM : out INTEGER) do
          ITEM := BUFFER(OUT_INDEX, ITEM_INDEX);
          end GET_ITEM;
          ITEM_INDEX := ITEM_INDEX mod BLOCK_LENGTH + 1;
          if ITEM_INDEX = 1 then
            -- a block has been consumed
            OUT_INDEX := OUT_INDEX mod BUFFER_SIZE + 1;
            BLOCK_COUNT := BLOCK_COUNT - 1;
          end if;
      or terminate; -- allows termination at end of block
    end select;
  end loop;
end BLOCK_TO_ITEM;
With use of the version of BLOCK TO ITEM just presented, we can restructure our example as follows, completely eliminating the use of abort.

procedure MAIN;
    task BLOCK TO ITEM is ... ;
    task PRODUCE BLOCK;
    task CONSUME ITEM;

    task body BLOCK TO ITEM is ... ;
    task body PRODUCE BLOCK is ... ;
    task body CONSUME ITEM is ... ;

begin -- body of MAIN
    null;
    -- await termination of tasks
end MAIN;
SELECT STATEMENT
Conditional Entry Calls

```
select
  entry call
  sequence_of_statements  -- optional
else
  sequence_of_statements
end select;
```

A conditional entry call issues an entry call if and only if this entry can be accepted immediately.
SELECT STATEMENT

Timed Entry Calls

select
entry call
sequence_of_statements -- optional
or
delay_statement
sequence_of_statements -- optional
end select;

A timed entry call issues an entry call if and only if this entry can be accepted within a given delay.
EXCEPTIONS IN TASKS

If an exception is raised in the sequence of statements of a task body that does not contain a handler for the exception, the execution of the task is abandoned; that is, the task is terminated. The exception is not propagated further.

Each task has an attribute named FAILURE which is an exception. Any task can raise the FAILURE exception in any task which it can name (for example T) by the statement

raise T'FAILURE;

The exception FAILURE supersedes any other exception that is not yet handled or that is received while handling FAILURE. Within the body of a task type T (and only there) there may be handlers for the exception T'FAILURE.
SUMMARY

Task Concepts

Entries

Accept Statements

Rendezvous

Task Attributes

Select Statements
CASE STUDY I

Program Design Using Packages
A TEXT FORMATTER

Default Operation

By default, output lines are filled and right justified
(by inserting extra spaces between words).

Line spacing is 1.

Right margin is set at column 60.

Page length is set at 66 with a four line margin
at the top and bottom of the page.

Leading spaces on a line cause a temporary indentation.

A blank line causes a break before it is transmitted to
the output. (A break terminates the current output
line in fill mode.)
### Command Summary

<table>
<thead>
<tr>
<th>command</th>
<th>break?</th>
<th>default</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bp</td>
<td>yes</td>
<td></td>
<td>begin page</td>
</tr>
<tr>
<td>.br</td>
<td>yes</td>
<td></td>
<td>cause a break</td>
</tr>
<tr>
<td>.ce n</td>
<td>yes</td>
<td>n=1</td>
<td>center next n lines</td>
</tr>
<tr>
<td>.fi</td>
<td>yes</td>
<td></td>
<td>start filling</td>
</tr>
<tr>
<td>.in n</td>
<td>no</td>
<td>n=0</td>
<td>indent n spaces</td>
</tr>
<tr>
<td>.ls n</td>
<td>no</td>
<td>n=1</td>
<td>line spacing is n</td>
</tr>
<tr>
<td>.nf</td>
<td>yes</td>
<td></td>
<td>stop filling</td>
</tr>
<tr>
<td>.pl n</td>
<td>no</td>
<td>n=66</td>
<td>set page length to n</td>
</tr>
<tr>
<td>.rm n</td>
<td>no</td>
<td>n=60</td>
<td>set right margin to n</td>
</tr>
<tr>
<td>.sp n</td>
<td>yes</td>
<td>n=1</td>
<td>space down n lines</td>
</tr>
<tr>
<td>.ti n</td>
<td>yes</td>
<td>n=0</td>
<td>temporary indent of n</td>
</tr>
</tbody>
</table>

A '.' in column 1 is an indication of a command line.

Signs are optional on command parameters; the presence of a sign indicates that the new value is relative to the old.
Main Program Design

procedure FORMAT
  begin
    Initialize
    while more input is available loop
      Get next line
      if line is a command then
        Process command
      else
        Process text
      end if
    end loop
  end loop
end FORMAT
procedure COMMAND

begin
  get parameter values (if any)
  case command type is
    when bp => break
      space to end of page
    when br => break
    when ce => break
      record number of lines to center
    when fi => break
      enter fill mode
    when in => set indent value
    when ls => set line space
    when nf => break
      enter no fill mode
    when pl => set page length
    when rm
    when sp => break
      space down n lines
    when ti => break
      set temp indent value
  end case
end COMMAND
Text Processing Design

procedure TEXT

begin

  handle leading blanks

  if line to be centered then
    align text
    put out line
  elsif line is blank then
    put out line
  elsif not in fill mode then
    put out line
  else -- handle word-by-word
    loop
      get a word
      exit when no more words
      put out word
    end loop
  end if

end TEXT
Collect subprograms which handle input and manipulate the input buffer into a package, with the buffer hidden within the body.

package INPUT_HANDLER is

    type COMMANDS is (BP, BR, CE, FI, IND, LS, NF, PL, RM, SP, TI, UNKNOWN);
    type SIGN_TYPE is (PLUS, MINUS, NONE, NO_PARAM);
    MAX_WORD_SIZE : constant INTEGER := 20;
    subtype WORD_STRING is STRING (1 .. MAX_WORD_SIZE);

    function READ_LINE return BOOLEAN;
        -- Reads a line into an internal buffer; returns
        -- FALSE when no more lines are available

    -- Command-related functions
    function IS COMMAND return BOOLEAN;
        -- TRUE if line starts with a ".."
    function COMMAND TYPE return COMMANDS
    procedure GET_VALUE (SIGN : out SIGN_TYPE;
        VALUE : out INTEGER);
        -- Reads parameters to commands, when present.

    -- Text processing functions
    procedure PROCESS_BLANKS;
        -- Handles leading blanks
    procedure CENTER;
    function BLANK_LINE return BOOLEAN;
    procedure NEXT_WORD (WORD : out WORD_STRING;
        LENGTH : out INTEGER);
    function LINE return STRING;
        -- used to send a whole line to FORMATTER
        -- after centering and leading blank removal.

end INPUT_HANDLER;
Collect subprograms which affect output into a single package. Output buffer and some status variables will be protected within the body of this package.

package FORMATTER is
    procedure BREAK;
    procedure SPACE (N : NATURAL);
        -- Space down N lines or to end of page.
    procedure PUTLINE (LINE : STRING);
        -- Used in no-fill mode
    procedure PUTWORD (WORD : STRING);
        -- Used in fill mode
end FORMATTER;
Use a package to hold values used in several places
(like a COMMON block).

package VALUES is
  FILL : BOOLEAN := TRUE;
  subtype VALUE_RANGE is
    INTEGER range 0 .. INTEGER'LAST;
  LINE_SPACING : VALUE_RANGE := 1;
  INDENT_VALUE, TEMP_INDENT, CENTER_COUNT :
    VALUE_RANGE := 0;
  RIGHT_MARGIN : VALUE_RANGE := 60;
  PAGE_LENGTH : VALUE_RANGE := 66;
end VALUES;
Implementation of FORMAT

with INPUT_HANDLER, VALUES, FORMATTER;
use INPUT_HANDLER, FORMATTER;
procedure FORMAT is
  -- main program
  procedure COMMAND is
    -- on following slide
  procedure TEXT is
    -- after COMMAND
  begin
    -- Initialization done in declarations
    while READ_LINE() loop
      if IS_COMMAND() then
        COMMAND;
      else
        TEXT;
      end if;
    end loop;
    -- Termination
    BREAK;
    SPACE(VALUES.PAGE_LENGTH); -- skip to end of page
  end FORMAT;
Within the procedure COMMAND, we will be changing some of the variables in VALUES. The nature of these changes will depend on the presence or absence of a sign on the parameter. Also, parameters themselves are optional. The following procedure will be used to uniformly handle the defaults and signs and with some appropriate checking.

procedure SET (VAR : in out VALUE_RANGE; -- one of the variables
               VAL : VALUE_RANGE; -- from the command line
               SIGN : SIGN_TYPE; -- from the command line
               DEFAULT : VALUE_RANGE := 0;
               MIN : VALUE_RANGE := 0; -- used for checking
               MAX : VALUE_RANGE := INTEGER'LAST) is

   begin

case SIGN is
    when NO_PARAM => VAR := DEFAULT;
    when PLUS    => VAR := VAR + VAL;
    when MINUS   => VAR := VAR - VAL;
    when NONE    => VAR := VAL;
end case;

   -- Check for illegal values
   if VAR > MAX then
      VAR := MAX;
   elsif VAR < MIN then
      VAR := MIN;
   end if;

   end SET;
Implementation of COMMAND
(within FORMAT)

with INPUT_HANDLER, VALUES, FORMATTER;
use INPUT_HANDLER, FORMATTER;
procedure FORMAT is
...

procedure COMMAND is
    subtype VALUE RANGE is VALUES.VALUE_RANGE;
    SIGN : SIGN_TYPE;
    VAL : VALUE_RANGE;
    SPACE_COUNT : INTEGER := 0;
    procedure SET
      ...
    end SET;
begin -- body of COMMAND
    GET VALUE (SIGN, VAL);
    case COMMAND_TYPE() is
      when BP => BREAK;
          SPACE (VALUES.PAGE_LENGTH);
      when BR => BREAK;
      when CE => BREAK;
          SET (VALUES.CENTER COUNT, VAL, SIGN, 1);
        -- note use of defaults
      when FI => BREAK;
          VALUES.FILL := TRUE;
      when IND => SET (VALUES.INDENT VALUE, VAL, SIGN);
          VALUES.TEMP.INDENT := VALUES.INDENT VALUE;
      when LS => SET (VALUES.LINE SPACING, VAL, SIGN, 1, 1);
      when NF => VALUES.FILL := FALSE;
      when PL => SET (VALUES.PAGE_LENGTH, VAL, SIGN, 66, 1);
      when RM => SET (VALUES.RIGHT_MARGIN, VAL, SIGN, 60, 1);
      when SP => BREAK;
          -- use SET to handle the sign and default
          SET (SPACE_COUNT, VAL, SIGN, 1);
          SPACE (SPACE_COUNT);
      when TI => BREAK;
          SET (VALUES.TEMP.INDENT, VAL, SIGN);
      when UNKNOWN => null; -- ignore
    end case;
    end COMMAND;
...
end FORMAT;
Implementation of TEXT
(within FORMAT)

with INPUT_HANDLER, VALUES, FORMATTER;
use INPUT_HANDLER, FORMATTER;

procedure FORMAT is

procedure TEXT is

   WORD : WORD_STRING;
   LENGTH : INTEGER;

   begin
   PROCESS_BLANKS;
   if VALUES.CENTER_COUNT > 0 then
      CENTER;
      PUTLINE (LINE());
      VALUES.CENTER_COUNT := VALUES.CENTER_COUNT - 1;
   elsif BLANK_LINE() or not VALUES.FILL then
      PUTLINE(LINE());
   else -- handle one word at a time
      loop
         NEXTWORD (WORD, LENGTH);
         exit when LENGTH = 0;
         PUTWORD (WORD(1..LENGTH));
      end loop;
   end if;
   end TEXT;

   end FORMAT;
package body INPUT_HANDLER is

MAX_LINE_LENGTH : constant INTEGER := 150;
BUFFER : STRING (1..MAX_LINE_LENGTH);
-- holds current input line
LENGTH, CURRENT : range 0..MAX_LINE_LENGTH;
-- LENGTH is length of current input line
-- CURRENT points into BUFFER when it is being
-- used word-by-word in fill mode.

function READ_LINE return BOOLEAN is
end READ_LINE;

function IS_COMMAND return BOOLEAN is
end IS_COMMAND;

function COMMAND_TYPE return COMMANDS is
end COMMAND_TYPE;

procedure GET_VALUE (SIGN : out SIGN TYPE;
VALUE : out INTEGER) is
end GET_VALUE;

procedure PROCESS_BLANKS is
end PROCESS_BLANKS;

procedure CENTER is
end CENTER;

function BLANK_LINE return BOOLEAN is
end BLANK_LINE;

procedure NEXT_WORDS (WORD : out WORD STRING;
LENGTH : out INTEGER) is
end NEXT_WORD;

function LINE return STRING is
end LINE;
end INPUT_HANDLER;
Design of GET_VALUE

procedure GET_VALUE (SIGN : out SIGN_TYPE;
VALUE : out INTEGER) is

begin

skip over command

skip intervening blanks

set SIGN

do conversion on characters to get VALUE

end
Implementation of GET_VALUE
(within INPUT_HANDLER)

package body INPUT_HANDLER is

MAX LINE LENGTH : constant INTEGER := 150;
BUFFER : STRING (1..MAX LINE LENGTH);
-- holds current input line
LENGTH, CURRENT : range 0..MAX LINE LENGTH;
-- LENGTH is length of current input line
-- CURRENT points into BUFFER when it is being
-- used word-by-word in fill mode.

procedure GET_VALUE (SIGN : out SIGN TYPE;
VALUE : out INTEGER) is
COL : range 1..MAX LINE_LENGTH;

function CONVERT (INDEX : INTEGER) return INTEGER is
-- converts a string of digits starting at INDEX in
-- BUFFER to an integer.
begin
-- Use the same technique as in RECORD_HANDLER.
end CONVERT;

begin
-- skip over command, three characters long
-- (could be generalized to handle arbitrary length
-- by looking for a special command syntax)
COL := 4;
SKIP_BLANKS(COL); -- skips blanks and tabs

if COL > LENGTH then
-- nothing left on line
SIGN := NO_PARAM;
VALUE := 0; -- should never be used, in this case
else
case BUFFER(COL) is
when '+' => SIGN := PLUS;
COL := COL + 1;
when '-' => SIGN := MINUS;
COL := COL + 1;
others => SIGN := NONE;
end case;
VALUE := CONVERT (COL);
-- CONVERT will convert a string of digits
-- starting at position COL to an INTEGER
end if;
end GET_VALUE;

end INPUT_HANDLER;
Implementation of INPUT_HANDLER

with VALUES, TEXT IO, FORMATTER;
use VALUES, TEXT IO, FORMATTER; -- FORMATTER needed for call to BREAK
package body INPUT_HANDLER is

MAX LINE_LENGTH : constant INTEGER := 150;
BUFFER : STRING (1..MAX LINE_LENGTH);
LENGTH, CURRENT : range 0..MAX_LINE_LENGTH;

function READLINE return BOOLEAN is
begin
if END OF FILE(STANDARD_INPUT) then
  return FALSE;
else
  LENGTH := 0;
  while not END OF LINE loop
    LENGTH := LENGTH + 1;
    GET(BUFFER(LENGTH));
  end loop;
  CURRENT := 1; -- used by NEXT_WORD
  return TRUE;
end if;
end READLINE;

function IS_COMMAND return BOOLEAN is
begin
  return BUFFER(1) = '.';
end IS_COMMAND;

function COMMAND_TYPE return COMMANDS is
FIRST : CHARACTER := BUFFER(2);
SECOND : CHARACTER := BUFFER(3);
C : COMMANDS;
begun
  C := UNKNOWN;
case FIRST is
when 'b' => if SECOND = 'p' then C := BP;
  elsif SECOND = 'r' then C := BR; end if;
when 'c' => if SECOND = 'e' then C := CE; end if;
when 'f' => if SECOND = 'i' then C := FI; end if;
when 'l' => if SECOND = 'n' then C := IND; end if;
when 'l' => if SECOND = 's' then C := LS; end if;
when 'n' => if SECOND = 'f' then C := NF; end if;
when 'p' => if SECOND = 'l' then C := PL; end if;
when 'r' => if SECOND = 'm' then C := RM; end if;
when 's' => if SECOND = 'p' then C := SP; end if;
when 't' => if SECOND = 'l' then C := TI; end if;
when others => null;
end case;
return C;
end COMMAND_TYPE;
procedure SKIP_BLANKS (I : in out INTEGER) is
  -- Advances I until BUFFER(I) is not a blank or tab.
  ...
end SKIP_BLANKS;

procedure GET_VALUE (SIGN : out SIGN_TYPE;
  VALUE : out INTEGER) is
  COL : range 1..MAX_LINE_LENGTH;
  ---------------------------------------------------------------
  function CONVERT (INDEX : INTEGER) return INTEGER is
    -- converts a string of digits starting at INDEX in
    -- BUFFER to an integer.
  begin
    -- Use the same technique as in RECORD_HANDLER.
    -- Return 0 if no digits encountered.
    ...
  end CONVERT;
  ---------------------------------------------------------------
  begin
    -- skip over command, three characters long
    -- (could be generalized to handle arbitrary length
    -- by looking for a special command syntax)
    COL := 4;
    SKIP_BLANKS(COL); -- skips blanks and tabs
    if COL > LENGTH then
      -- nothing left on line
      SIGN := NO_PARAM;
      VALUE := 0; -- should never be used, in this case
    else
      case BUFFER(COL) is
        when '+' => SIGN := PLUS;
        COL := COL + 1;
        when '-' => SIGN := MINUS;
        COL := COL + 1;
        others => SIGN := NONE;
      end case;
      VALUE := CONVERT (COL);
      -- CONVERT will convert a string of digits
      -- starting at position COL to an INTEGER
    end if;
  end GET_VALUE;
procedure PROCESS_BLANKS is
  -- Remove leading blanks, incrementing temporary indent
  -- counter appropriately.
  NUM_BLANKS : range 0..MAX_LINE_LENGTH;
begin
  if BUFFER(1) /= ' ' then
    return; -- This procedure is not relevant.
  end if;
  BREAK; -- .ti causes a break
  -- Find first non-blank;
  NUM_BLANKS := 1;
  while NUM_BLANKS < LENGTH
    and then BUFFER(NUM_BLANKS+1) = ' ' loop
    NUM_BLANKS := NUM_BLANKS + 1;
  end loop;
  -- Process result
  if NUM_BLANKS = LENGTH then
    LENGTH := 0; -- indication of a blank line
  else
    TEMP_INDENT := NUM_BLANKS + INDENT_VALUE;
    BUFFER(1..LENGTH-NUM_BLANKS)
    := BUFFER(NUM_BLANKS+1..LENGTH);
    LENGTH := LENGTH - NUM_BLANKS;
  end if;
end PROCESS_BLANKS;

procedure CENTER is
  -- Centering is accomplished by manipulation of TEMP_INDENT.
  NEW-VALUE : INTEGER;
begin
  NEW_VALUE := (RIGHT_MARGIN + TEMP_INDENT - LENGTH) / 2;
  if NEW_VALUE > 0 THEN
    TEMP_INDENT := NEW_VALUE;
  end if;
end CENTER;

function BLANK_LINE return BOOLEAN is
begin
  return LENGTH = 0;
end BLANK_LINE;

function LINE return STRING is
begin
  return BUFFER(1..LENGTH);
end LINE;
Implementation of INPUT_HANDLER
(Continued)

procedure NEXT_WORD (WORD : out WORD_STRING;
LENGTH : out INTEGER) is
  -- Uses the variable CURRENT. LENGTH will tell how many
  -- characters in WORD are significant. Any string of
  -- non-blank characters is a 'word'.
end NEXT_WORD;
end INPUT_HANDLER;
Outline of FORMATTER

package body FORMATTER is

MAX_LINE_LENGTH : constant INTEGER := 132;
MARGIN : constant INTEGER := 4;
BUFFER : STRING (1..MAX_LINE_LENGTH);
-- Current output line
OUT_PTR, OUT_WORDS, LINE_NUM : VALUE RANGE := 0;
-- OUT_PTR points to last character in BUFFER
-- OUT_WORDS is the number of words on this line
-- LINE_NUM is the current line number

procedure BREAK is
...
end BREAK;

procedure SPACE (N : NATURAL) is
...
end SPACE;

procedure PUTLINE (LINE : STRING) is
...
end PUTLINE;

procedure PUTWORD (WORD : STRING) is
...
end PUTWORD;

end FORMATTER;
with VALUES, TEXT IO;
use VALUES, TEXT IO;
package body FORMATTER is

MAX_LINE_LENGTH : constant INTEGER := 132;
BUFFER : STRING (1..MAX_LINE_LENGTH);
OUT_PTR, OUT_WORDS, LINE_NUM : VALUE_RANGE := 0;
MARGIN : constant INTEGER := 4;
BLANK : constant CHARACTER := ' ';
BOTTOM : constant INTEGER := PAGE_LENGTH - MARGIN;

function MIN (I, J : INTEGER) return INTEGER is
begin
  if I < J then
    return I;
  else
    return J;
  end if;
end MIN;

procedure PUTLINE (LINE : STRING) is
  -- Send LINE to the output file
  BLANKS : constant STRING := (1..MAX_LINE_LENGTH => BLANK);
begin
  if LINE_NUM = 0 or LINE_NUM > BOTTOM then
    -- start a new page
    NEW LINE (MARGIN); -- puts out blank lines
    LINE_NUM := MARGIN + 1;
  end if;
  -- put out leading blanks
  PUT (BLANKS(1..TEMP_INDENT));
  TEMP_INDENT := INDENT VALUE;
  -- write out the string LINE
  PUT (LINE);
  -- handle line spacing
  NEW LINE (MIN (LINE_SPACING, BOTTOM-LINE_NUM+1));
  LINE_NUM := LINE_NUM + LINE_SPACING;
  -- check for end of page
  if LINE_NUM > BOTTOM then
    NEW LINE (MARGIN);
    -- LINE_NUM is purposely not changed here
  end if;
end PUTLINE;
Implementation of FORMATTER
(Continued)

procedure SPACE (N : NATURAL) is
  -- skip N lines or to bottom of page
begin
  if LINE_NUM > BOTTOM then
    -- spacing has no effect in this case
    return;
  end if;
  if LINE_NUM = 0 then
    NEW_LINE (MARGIN);
    LINE_NUM := MARGIN + 1;
  end if;
  NEW_LINE (MIN (N, BOTTOM-LINE_NUM+1));
  LINE_NUM := LINE_NUM + N;
  -- check for end-of-page
  if LINE_NUM > BOTTOM then
    NEW_LINE (MARGIN);
  end if;
end SPACE;

procedure BREAK is
  -- end current filled line
begin
  if OUT_PTR > 0 then
    PUTLINE(BUFFER(1..OUT_PTR));
    OUT_PTR := 0;
    OUT_WORDS := 0;
  end if;
end BREAK;

procedure PUTWORD (WORD : STRING) is
...
end PUTWORD;
end FORMATTER;
Design of PUTWORD

procedure PUTWORD

begin

  Compute current line length + word length

  if new length > allowed line length then

    -- Addition of blanks necessary to right-justify
    Spread out words in buffer to fill line
    Break -- to flush out the line

  end if

  Copy word to output buffer

  Adjust state variables

end PUTWORD;
Design of SPREAD

procedure SPREAD

-- the number of blanks to add will be passed as a parameter

begin

Switch direction flag

-- add blanks from opposite ends on alternate lines

Compute number of holes -- spaces between words

loop from end to beginning of words in buffer

copy a character to next available slot

if character is a blank then

insert appropriate number of extra blanks

-- based on number of holes

end if

end loop

end SPREAD
Implementation of PUTWORD
(within FORMATTER)

package body FORMATTER is

MAX_LINE_LENGTH : constant INTEGER := 132;
MARGIN : constant INTEGER := 4;
BUFFER : STRING (1..MAX_LINE_LENGTH);

-- Current output line
OUT_PTR, OUT_WORDS, LINE_NUM : VALUE_RANGE := 0;
-- OUT_PTR points to last character in BUFFER
-- OUT_WORDS is the number of words on this line
-- LINE_NUM is the current line number

...

procedure PUTWORD (WORD : STRING) is
  LAST, LINE_SIZE : VALUE_RANGE;
  begin
  LINE_SIZE := RIGHT_MARGIN - TEMP_INDENT;
  if OUT_PTR + WORD'LENGTH > LINE_SIZE then
    -- Addition of blanks necessary to right-justify
    SPREAD (LINE_SIZE - OUT_PTR + 1);
    -- "+ 1" because BUFFER(OUT_PTR) is a blank
    if OUT_WORDS > 1 then
      OUT_PTR := LINE_SIZE; -- the effect of SPREAD
      end if;
    BREAK;
  end if;
  -- Copy WORD and a blank to output buffer
  LAST := OUT_PTR + WORD'LENGTH + 1;
  BUFFER(OUT_PTR+1..LAST) := WORD & BLANK;
  -- Adjust state variables
  OUT_PTR := LAST;
  OUT_WORDS := OUT_WORDS + 1;
  end PUTWORD;

...

end FORMATTER;
Implementation of SPREAD  
(within PUTWORD)

package body FORMATTER is

MAX_LINE_LENGTH : constant INTEGER := 132;
MARGIN : constant INTEGER := 4;
BUFFER : STRING (1..MAX_LINE_LENGTH);
-- Current output line
OUT_PTR, OUT_WORDS, LINE_NUM : VALUE RANGE := 0;
-- OUT_PTR points to last character in BUFFER
-- OUT_WORDS is the number of words on this line
-- LINE_NUM is the current line number

ADD_FROM_RIGHT : BOOLEAN := TRUE;
-- must be at the package body level; used by SPREAD to
-- insert blanks at opposite ends of alternate lines

procedure PUTWORD (WORD : STRING) is

procedure SPREAD (NUM_BLANKS : VALUE RANGE) is
I, J, NUM_HOLES, ADD_COUNT : VALUE RANGE;
NUM_EXTRA: VALUE RANGE := NUM_BLANKS;
begin
if OUT_WORDS <= 1 then
return; -- nowhere to put blanks
end if;
ADD_FROM_RIGHT := not ADD_FROM_RIGHT;
-- add blanks from opposite ends on alternate lines
NUM_HOLES := OUT_WORDS - 1;
I := OUT_PTR - 1; -- points to last non-blank char
J := I + NUM_EXTRA;
while I < J loop
BUFFER(J) := BUFFER(I);
if BUFFER(J) = BLANK then
if ADD_FROM_RIGHT then
ADD_COUNT := (NUM_EXTRA - 1) / NUM_HOLES + 1;
else
ADD_COUNT := NUM_EXTRA / NUM_HOLES;
end if;
NUM_EXTRA := NUM_EXTRA - ADD_COUNT;
NUM_HOLES := NUM_HOLES - 1;
for K in 1..ADD_COUNT loop
J := J - 1;
BUFFER(J) := BLANK;
end loop;
end if;
end loop;
end SPREAD;

end PUTWORD;

end FORMATTER;

CSI.360
CASE STUDY II

TELEPHONE SWITCHING SIMULATION
System Block Diagram

LINE HANDLERS

CALL PROCESSOR

\[\text{to telephones}\]
Network Operation

Each line handler monitors its associated telephone lines for such events as digits being transmitted and the receiver being lifted from or returned to the hook. When these events occur, the line handler notifies the call processor. Upon command from the call processor, it also controls ringing. The line handlers are used (rather than a single central processor) in order to distribute the real-time demands of line monitoring.

The call processor is driven by messages from the line handlers concerning line events. It translates phone numbers to physical line addresses and controls the connection and disconnection of circuits.

This simulation will only be concerned with the transmission of control signals among the various components of the network and the interpretation of these signals. Data could be collected to determine the adequacy of the components and the architecture of the network to handle various traffic loads.
Program Task Structure

The following tasks will exist throughout the execution of the simulation:

- The CALL PROCESSOR will be represented by a task.
- Each LINE HANDLER will be represented by an identical task.
- Each telephone will be represented by a PHONE task.
- Calls will be generated by a DRIVER task.

Each call will be represented by a dynamically allocated CALL task, which will communicate with the PHONE tasks involved. Such tasks will terminate when the calls they represent are completed.

The control signals flowing through the network will be represented by messages passed among these tasks.
MESSAGE

A single message type will be useful, so that all message handling can be done uniformly. We will use the following declarations to define such a message type.

type MSG_TYPE is (NOISE, DIGIT, HOOK, STATUS, DETAIL);

type STATUS_TYPE is (RINGING, BUSY, DIALTONE, CONNECTED, DISCONNECTED, COMPLETED, NOANSWER, PHONEFREE, NOTFREE);

type MESSAGE (KIND : MSG_TYPE) is
record
  SENDER : INTEGER; -- to identify source
  LINE_NUM : INTEGER; -- sometimes needed
  case KIND is
    when NOISE => RING : BOOLEAN;
      -- start phone ringing if TRUE
      -- stop if FALSE
    when DIGIT => DIGIT : INTEGER;
    when HOOK => HOOK_STATE : (ON, OFF);
    when STATUS => STXTE : STATUS_TYPE;
    when DETAIL => LENGTH : INTEGER; -- length of call
      FROM : INTEGER; -- calling line number
      TO : INTEGER; -- number being called
      HANGUP : INTEGER; -- which one hangs up
  end case;
end MESSAGE;
Communication between Tasks

We want to send messages between tasks asynchronously so that, for example, a LINE HANDLER need not wait until the CALL PROCESSOR has actually processed one of its messages before it can receive a message from a PHONE. We will thus need tasks to handle the mechanics of message buffering. Each task will have a corresponding message buffer task to handle its incoming communication.

```
task type MESSAGE_BUFFER is
    entry SEND (M : in MESSAGE);
      -- called by other tasks to send a message to the
      -- corresponding task
    entry RECEIVE (M : out MESSAGE);
      -- called by the corresponding task to accept messages
end MESSAGE_BUFFER;
```

Since MESSAGE is a globally declared record type with variants to represent all of the different kinds of messages which might be used by any of the tasks, we need only write one message buffering task.
Simulation Primitives

To implement a simulation capability, we need routines to maintain an event list, to keep track of a simulation time and to allow tasks to be scheduled for execution. In this particular problem, the only scheduling primitive needed by the tasks representing the various system components is hold, which allows a given task to suspend its execution for a fixed amount of simulation time.

The simulation routines will be implemented as a package. Any tasks wishing to use hold must have previously been assigned a task identifier by the simulation package. A procedure will be available in the package for this package.

package SIMULATION is
  type TASK_ID is private;
  procedure GET_ID (ID : out TASK_ID);
  -- used to ask for a task identifier
  procedure RETURN_ID (ID : in TASK_ID);
  -- used by dynamic process when they terminate
  procedure HOLD (ID : in TASK_ID; TIME : in INTEGER);
  -- TIME is milliseconds of simulation time
  procedure RECEIVE_MESSAGE (BUFFER : in MESSAGE_BUFFER;
                             M : out MESSAGE);
  -- called by a task when it wants to remove a
  -- message from its buffer
private
  type TASK_ID is new INTEGER;
end SIMULATION;

The RECEIVE MESSAGE procedure is necessary in order to allow the simulation package to know about those tasks which are suspended waiting for message, as well as those suspended by calls to hold.
Main Program Structure

procedure SWITCH (NUM_LINES : INTEGER; -- not greater than 8999
RUN_LENGTH : INTEGER) -- simulation time

is

-- message declarations (as on earlier slide) go here


task type MESSAGE_BUFFER is
  entry SEND (M : in MESSAGE);
  entry RECEIVE (M : out MESSAGE);
end MESSAGE_BUFFER;

package SIMULATION is

  -- as on previous slide

end SIMULATION

task CALL_PROCESSOR;

task type LINE_HANDLER is
  entry STARTUP (INDEX : INTEGER);
end LINE_HANDLER;

task type PHONE is
  entry STARTUP (INDEX : INTEGER);
end PHONE;

task type CALL; -- these are allocated dynamically

task DRIVER; -- generates calls
Main Program (continued)

-- declarations of constants and variables

MAX_LINE_NUM : constant INTEGER := NUM_LINES - 1;

MAX_HANDLER : constant INTEGER := MAX_LINE_NUM / 10 + 1;
-- maximum of ten lines per handler

-- Phone numbers will be represented by four digits.
-- The first three digits minus 100 will be the handler number.
-- The fourth digit will be the line number belonging to
-- that handler. The smallest phone number is 1000,
-- corresponding to line 0 of handler 000.

HANDLERS : array (0..MAX_HANDLER) of LINE_HANDLER;
HANDLER_BUFFERS : array(0..MAX_HANDLER) of MESSAGE_BUFFER;

PHONES : array (0..MAX_LINE_NUM) of PHONE;
PHONE_BUFFERS : array(0..MAX_LINE_NUM) of MESSAGE_BUFFER;

PROCESSOR_BUFFER : MESSAGE_BUFFER;
DRIVER_BUFFER : MESSAGE_BUFFER;

use SIMULATION; -- needed in main program body

MAIN_TASK : TASK_ID;

-- Bodies of tasks and the SIMULATION package would go here


Main Program (continued)

begin -- body of SWITCH

-- send buffer indices to line handler and call receiver tasks
for I in 0..MAX_LINE_NUM loop
   PHONES(I).STARTUP (INDEX => I);
end loop;

for I in 0..MAX_HANDLER loop
   HANDLERS(I).STARTUP (INDEX => I);
end loop;

-- wait for RUN_LENGTH simulation time to elapse
GET ID (MAIN_TASK);
HOLD (MAIN_TASK, RUN_LENGTH);

-- Produce statistics and terminate all tasks
   :
   :
   :

end SWITCH;
Body of MESSAGE_HANDLER

task body MESSAGE_HANDLER is
  -- We will assume the availability of a generic package
  -- called LINKED_LIST, which is much like SORTED_LIST
  -- except that there are no priorities involved and
  -- insert puts the new item at the end of the list.

  package MESSAGE_LIST is new LINKED_LIST(MESSAGE);
  use MESSAGE_LIST;

  MESSAGES : LIST;
  COUNT : INTEGER := 0;

begin

  CREATE (MESSAGES);

  loop -- no exit from this loop except by termination
    select
      when COUNT > 0 =>
        accept RECEIVE (M : out MESSAGE) DO
          NEXT ENTRY (MESSAGES, M);
          COUNT := COUNT - 1;
        end RECEIVE;
      or accept SEND (M : in MESSAGE) do
        INSERT (MESSAGES, M);
        COUNT := COUNT + 1;
      end SEND;
      or when COUNT = 0 => terminate;
    end select;
  end loop;

end MESSAGE_BUFFER;
package body SIMULATION is

-- Since the event list is a shared data structure, a task will be
-- used to synchronize access to it.
task LIST_HANDLER is
  entry ADD ENTRY (ID : TASK_ID; TIME : INTEGER);
  entry ADVANCE_TIME;
end LIST_HANDLER;

-- A task will be used to manage task ids, again because of
-- shared data structures;
task ID_MANAGER is
  entry GET_ID (ID : out TASK_ID);
  entry RETURN_ID (ID : in TASK_ID);
end ID_MANAGER;

-- A task will be necessary to keep count of the number of
-- tasks suspended, in order to know when to advance the
-- simulation time.
task COUNTER is
  entry INCREMENT;
  entry DECREMENT;
  entry INCREMENT_TOTAL;
  entry DECREMENT_TOTAL;
end COUNTER;

-- A task type is introduced to implement task suspension.
task type SIGNAL is
  entry SEND;
  entry WAIT;
end SIGNAL;

MAX_TASK_ID : constant TASK_ID := MAX_LINE_NUM * 2;

SIGNS: array (1..MAX_TASK_ID) of SIGNAL;

-- one for each task which could be suspended

task body SIGNAL is
begin
  loop
    accept SEND;
    accept WAIT;
  end loop;
end SIGNAL;
procedure GET_ID (ID : out TASK_ID) is
begin
  ID_MANAGER.GET_ID (ID);
  COUNTER.INCREMENT_TOTAL;
end GET_ID;

procedure RETURN_ID (ID : in TASK_ID) is
begin
  ID_MANAGER.RETURN_ID (ID);
  COUNTER.DECREMENT_TOTAL;
end RETURN_ID;

procedure HOLD (ID : TASK_ID; TIME : INTEGER) is
begin
  LIST_HANDLER.ADD_ENTRY (ID, TIME);
  COUNTER.INCREMENT;
  SIGNALS(ID).WAIT; -- suspends this procedure until
  COUNTER.DECREMENT;
end HOLD;

procedure RECEIVE_MESSAGE (BUFFER : in MESSAGE_BUFFER;
                            M : out MESSAGE) is
begin
  select
    BUFFER.RECEIVE (M);
  else -- no messages currently available
    COUNTER.INCREMENT;
    BUFFER.RECEIVE (M);
    -- will cause suspension until a message comes
    COUNTER.DECREMENT;
  end select;
end RECEIVE_MESSAGE;

CSII.220
task body LIST_HANDLER is
-- This task will use a package like SORTED LIST to implement
-- an event list, except that the items must be sorted in
-- ascending priority order.
-- (The "priorities" are event times.)

package LIST_PACKAGE is new ASCENDING_SORTED_LIST (TASK_ID);
use LIST_PACKAGE;

EVENT_LIST : LIST:
  ID : TASK_ID;
  SIM_TIME : INTEGER := 0; -- simulation time
begin
  CREATE (EVENT_LIST);
  loop
    select
      accept ADD_ENTRY (ID : TASK_ID; TIME : INTEGER) do
        INSERT (EVENT_LIST, ID, SIM_TIME+TIME);
      end ADD_ENTRY;
    or accept ADVANCE TIME;
        NEXT ENTRY (EVENT_LIST, ID, SIM_TIME);
      SIGNALS(ID).SEND; -- awakens a Task in HOLD
    end select;
  end loop;
end LIST_HANDLER;

task body ID_MANAGER is
  NEXT_TASK_ID : INTEGER := 0;
  ID_SET : Array (1..MAX_TASK_ID) of range 0..MAX_TASK_ID;
begin
  for I in 1..MAX_TASK_ID loop
    ID_SET(I) := I+1;
  end loop;
  ID_SET(MAX_TASK_ID) := 0;
  loop
    select
      when NEXT_TASK_ID /= 0
        accept GET_ID (ID : out TASK_ID) do
          ID := NEXT_TASK_ID;
          NEXT_TASK_ID := ID_SET(NEXT_TASK_ID);
        end GET_ID;
      or accept RETURN_ID (ID : in TASK_ID) do
        ID_SET(ID) := NEXT_TASK_ID;
        NEXT_TASK_ID := ID;
      end RETURN_ID;
    end select;
  end loop;
end ID_MANAGER;
SIMULATION (continued)

task body COUNTER is
  TOTAL_TASKS, SUSPENDED_TASKS : INTEGER := 0;
begin
  loop
    select
      accept INCREMENT_TOTAL do
        TOTAL_TASKS := TOTAL_TASKS + 1;
      end INCREMENT_TOTAL;

      or accept DECREMENT_TOTAL do
        TOTAL_TASKS := TOTAL_TASKS - 1;
      end DECREMENT_TOTAL;

      or accept INCREMENT do
        SUSPENDED_TASKS := SUSPENDED_TASKS + 1;
        if SUSPENDED_TASKS >= TOTAL_TASKS then
          ADVANCE_TIME;
        end if;
      end INCREMENT;

      or accept DECREMENT do
        SUSPENDED_TASKS := SUSPENDED_TASKS - 1;
      end DECREMENT;

      or terminate;
    end select;
  end loop;
end COUNTER;
end SIMULATION;
Body of LINE_HANDLER;

The following task body provides a simple example of the use of the simulation and message buffering capabilities by a task which represents one of the simulation objects.

task body LINE_HANDLER is

M : MESSAGE;
MY_NUMBER : INTEGER; -- used as message buffer index
ME^ : TASK_ID; -- for identification to SIMULATION package

HANDLING_TIME : constant := 50; -- units of simulation time

use SIMULATION;

begin
accept STARTUP (INDEX : INTEGER) do
MY_NUMBER := INDEX;
end STARTUP;

GET_ID (ME);

loop -- loops forever, simulating a line handler

RECEIVE MESSAGE (HANDLER_BUFFERS(MY_NUMBER), M);

case M.KIND is
when DIGIT | HOOK =>
-- line event; pass on to call processor
M.SENDER := MY_NUMBER;
PROCESSOR_BUFFER.SEND (M);

when STATUS | NOISE =>
-- from call processor; send on to phone
M.SENDER := MY_NUMBER;
PHONE_BUFFERS(M.LINE_NUM).SEND (M);

when DETAIL => null; -- should never occur
end case;

-- simulate processor time used to handle message
HOLD (ME, HANDLING_TIME);
end loop;

end LINE_HANDLER;
SUMMARY

SYNTAX
- designed for readability

DECLARATIONS and TYPES
- factorization of properties, maintainability
- abstraction, hiding of implementation details
- reliability, due to checking
- floating point and fixed point, portability
- access types, utility and security

STATEMENTS
- assignment, iteration, selection, transfer
- uniformity of syntax (comb structure)
- generally as simple as possible
  (e.g., iteration control)

SUB. GRAMS
- procedures and functions
- logically described parameter modes
  (as opposed to definition by implementation description)
- overloading

PACKAGES
- modularity and abstraction
- structuring for complex programs
- hiding of implementation, maintainability
- major uses:
  . named collections of declarations
  . groups of related subprograms
  . encapsulated data types
LIBRARIES
- separate compilation
- generics
- program development environment

TASKING
- can be done completely with Ada features
- single concept for intertask communication and synchronization
- interface with external devices
- designed for efficient implementation

EXCEPTION HANDLING
- for reliability of real-time systems
- standard vs. user-defined exceptions
- meant mainly for handling errors (rather than as a general programming technique)

MACHINE DEPENDENCIES
- representation specifications
- interface with other languages
- low level I/O
Ada IS DESIGNED FOR WRITING LARGE PROGRAMS

Ada HAS FEATURES TO ALLOW SUITABLE EXTENSIONS FOR A PARTICULAR APPLICATION

Ada IS A DESIGN LANGUAGE
What haven't we discussed ???

GO TO statements

Representation Specifications

Details of Generics

Input-Output

Pragmas

Inline procedures

Interface to other languages
HELBAT BIFF

HUMAN
ENGINEERING
LABORATORIES
BATTALION
ARTILLERY
TEST

BATTLEFIELD
IDENTIFICATION
FRIEND
OR
FOE
PROBLEM STATEMENT

FIRE AT (AND HIT) ENEMY TARGETS

FUNCTIONAL SPECIFICATION (PARTIAL)

INPUT FROM - RADAR UNIT
HUMAN OPERATOR

OUTPUT TO - HUMAN OPERATOR
REMOTE ARTILLERY
LOCAL WEAPON CONTROL

OPERATOR DISPLAY - Plasma Scope (nominally 9260 baud)

OPERATOR INPUT DEVICE - Touch Panel
### RADAR INPUT

DMA (DIRECT MEMORY ACCESS) DUMP. EVERY 20 MILLISECONDS
ON INTERRUPT FROM RADAR HARDWARE. OF 19 16-BIT "WORDS".

**FORMAT:**

<table>
<thead>
<tr>
<th>WORD(S)</th>
<th>BIT(S)</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 .. 13</td>
<td>ANTENNA AZIMUTH</td>
</tr>
<tr>
<td>1</td>
<td>0 .. 1</td>
<td>1-ND BEACON ID</td>
</tr>
<tr>
<td>1</td>
<td>2 .. 13</td>
<td>1-ND BEACON RANGE</td>
</tr>
<tr>
<td>2</td>
<td>0 .. 1</td>
<td>2-ND BEACON ID</td>
</tr>
<tr>
<td>2</td>
<td>2 .. 13</td>
<td>2-ND BEACON RANGE</td>
</tr>
<tr>
<td>3</td>
<td>0 .. 13</td>
<td>CENTER OF SCAN SECTOR</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>IN INTERROGATE MODE?</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>SEARCH RANGE (SHORT, LONG)</td>
</tr>
<tr>
<td>4</td>
<td>2 .. 3</td>
<td>WIDTH OF SCAN SECTOR</td>
</tr>
<tr>
<td>4</td>
<td>4 .. 5</td>
<td>DIRECTION OF SCAN</td>
</tr>
<tr>
<td>4</td>
<td>6 .. 7</td>
<td>RATE OF SCAN</td>
</tr>
<tr>
<td>5 .. 17</td>
<td>0 .. 199</td>
<td>RANGE PROFILE</td>
</tr>
<tr>
<td>18</td>
<td>0 .. 15</td>
<td>ERROR_FLAG</td>
</tr>
</tbody>
</table>

CSIII.030
POLICY - destroy enemy targets

locate a target -

if it's not friendly, then destroy it
Simplified Actor Model

**PERCEPTOR**
perception of external and internal states

**PROCESSOR**
decide on basis of policy and perception what action to take

**EFFECTOR**
cause change in external or internal states
Simplified Actor Model
Simplified Actor Model

- **Perceptor**: Perception of external and internal states
- **Processor**: Decide on basis of policy and perception what action to take
- **Effector**: Cause change in external or internal states
Simplified Actor Model

PERCEPTOR
perception of external and internal states

PROCESSOR
deide on basis of policy and perception what action to take

EFFECCTOR
cause change in external or internal states

POLICY
takes input from environment and determines policy

Environment

Internal states

Actor Model

Environment
PROCESSOR IMPLEMENTATION

FOR THIS SYSTEM:   HUMAN DECISION MAKER

ATTRIBUTES:

INPUT   - INFORMATION RATE? PERCEIVABLE STIMULI?
          . . .

OUTPUT   - INFORMATION RATE? MODES (HANDS, VOICE, ...)

SYSTEM   - ALERTNESS
          RESPONSE TIME
          PROFICIENCY

(EMBEDDED HUMAN SYSTEM)
PERCEPTOR
IMPLEMENTATION

( perceptor )
RADAR

( processor )
DISPLAY
PREPARATION

( effector )
PLASMA
SCOPE

environment

environment
DISPLAY
PREPARATION

(environment) ->

(perceptor)
relate radar information to display
relate operator actions to display

(processor)
turn display information into commands for Plasma Scope
(buffer)

(effector)
send commands to Plasma Scope

(environment) ->
EFFECCTOR
IMPLEMENTATION

(perceptor)

TOUCH

PANEL

(processor)

Interpret info from touch panel,
choose appropriate operations

(effector)

WEAPON

TRANSMITTER

(operator's display)

environment

environment
INTERPRETER IMPLEMENTATION

(perceptor) -> (processor) -> (effector)

environment

touch panel interface -> command dispatcher

weapon control interface
transmitter interface
operator display interface

environment

environment
HELBAT BIFF

PERCEPTOR - (RADAR)
  SENSOR - (RADAR)
  OPERATOR'S DISPLAY HANDLER
  DISPLAY DEVICE COMMAND FORMATTER
  BUFFER
  DISPLAY DEVICE WRITER
  ENVIRONMENTAL SENSOR INFORMATION
  SENSOR INTERFACE
  SENSOR INFORMATION DISPLAY GENERATOR
  INTERNAL INFORMATION FROM OPERATOR COMMANDS
  DISPLAY DEVICE - (PLASMA SCOPE)

PROCESSOR - (HUMAN OPERATOR)

EFFECTOR -
  OPERATOR INPUT DEVICE - (TOUCH PANEL)
  OPERATOR COMMAND HANDLER
  COMMAND DISPATCHER
  OPERATOR INPUT DEVICE READER
  DISPLAY AND EFFECTOR CONTROL
  CURSOR AIMING
  COMMAND INDICATOR LIGHTING
  WEAPON AIMING
  TARGET LOCATION TRANSMISSION HANDLER
  WEAPON TRANSMITTER
  OPERATOR'S DISPLAY HANDLER

CSIII.180
with Linked_List_FIFO_Queue, Ring_Queue;

procedure HELBAT_BIFF is

package Common_Definitions is

end Common_Definitions;

package Operator_Display_Handler is

package Display_Device_Command.Formatter is

package Display_Device_Command_Buffer is new Ring_Queue ( ... );

-- declarations of procedures that handle
-- coding and buffering of commands for
-- other tasks

end Display_Device_Command_Formatter;

-----------------------------

task type Display_Device_Writer;

-----------------------------

package Sensor_Information is

package Sensor_Definitions is

end Sensor_Definitions;

-----------------------------

task type Sensor_INTERFACE is

-- declarations of entries and
-- representation specification
end Sensor_INTERFACE;

-----------------------------

task type Sensor_Information_Display_Generator;

end Sensor_Information;

end Operator_Display_Handler;

-----------------------------

CSIII.190
package Operator_Command_Handler is

package Operator_Command_Definitions is
end Operator_Command_Definitions;

--

task type Command_Dispatcher is
end Command_Dispatcher;

--

task type Operator_Input_Device_Reader;

--

package Display_and_Effector_Control is

package Aiming_Information is
end Aiming_Information;

--

task type Aiming_Cursor_Operations;
task type Command_Indicator_Lighting;
task type Weapon_Aiming;
task type Target_Location_Transmission_Handler;
end Display_and_Effector_Control;
end Operator_Command_Handler;

--

-- package bodies are separately compiled
...

type Display_Writer is access
  Operator_Display_Handler.Display_Device_Writer;
-- note: this type points to tasks
...

Plasma_Scope_Writer : Display_Writer;
...

begin    -- body of HELBAT_BIFF
...

CSIII.200
BEGIN -- HELBAT_BIFF

LOOP

BEGIN -- ACTIVATE TASKS IN PROPER ORDER

...

DELAY 10 * SECONDS;
Plasma_Scope_Writer := new Display_Writer;

...

END;

END LOOP;

END HELBAT_BIFF:
package Sensor_Definitions is

   FOURTEEN_BITS_FULL : constant Integer := 16#3FFF#;

   subtype RA2 is Integer range 0..FOURTEEN_BITS_FULL;

   subtype Range_Bin is Integer range 0..199;

   type Direction is (NONE, LEFT_TO_RIGHT, RIGHT_TO_LEFT,
                       SEARCH_LIGHT);                  

   for Direction use (NONE         => 0,
                       LEFT_TO_RIGHT => 1,
                       RIGHT_TO_LEFT => 2,
                       SEARCH_LIGHT => 3);

   type Profile_Of_Range is 
      array ( Range_Bin'FIRST .. Range_Bin'LAST ) of Boolean;

   type Radar_Input is
      record
         Antenna_Azimuth    : RA2;
         First_Beacon_ID   : Integer range 0..3;
         First_Beacon_Range: Integer range 0..4095;
         Second_Beacon_ID  : Integer range 0..3;
         Second_Beacon_Range: Integer range 0..4095;
         Center_of_Scan_Sector: RA2;
         In_Interrogate_Mode: Boolean;
         First_Beacon_Range: Integer range 0..1;
         Second_Beacon_Range: Integer range 0..3;
         Search_Range: Integer range 0..3;
         Width_of_Scan_Sector: Integer range 0..3;
         Direction_of_Scan: Direction;
         Rate_of_Scan: Integer range 0..3;
         Range_Profile: Profile_Of_Range;
         Error_Flag : Integer range 0..16#FFFF#;
      end record;

end Sensor_Definitions;
package Sensor_Definitions is

   FOURTEEN_BITS_FULL : constant Integer := 16#3FFF#

   subtype Raz is Integer range 0..FOURTEEN_BITS_FULL;

   subtype Range_Bin is Integer range 0..199;

   type Direction is (NONE, LEFT_TO_RIGHT, RIGHT_TO_LEFT, SEARCH_LIGHT);

   for Direction use (NONE => 0,
      LEFT_TO_RIGHT => 1,
      RIGHT_TO_LEFT => 2,
      SEARCH_LIGHT => 3);

   type Profile_Of_Range is
      array ( Range_Bin'FIRST .. Range_Bin'LAST ) of Boolean;

   type Radar_Input is
      record
         Antenna_Azimuth : Raz;
         First_Beacon_ID : Integer range 0..3;
         First_Beacon_Range : Integer range 0..4095;
         Second_Beacon_ID : Integer range 0..3;
         Second_Beacon_Range : Integer range 0..4095;
         Center_of_Scan_Sector : Raz;
         In_Interrogate_Mode : Boolean;
         Search_Range : Integer range 0..1;
         Width_of_Scan_Sector : Integer range 0..3;
         Direction_of_Scan : Direction;
         Rate_of_Scan : Integer range 0..3;
         Range_Profile : Profile_Of_Range;
         Error_Flag : Integer range 0..16#FFFF#;
      end record;

end Sensor_Definitions;
-- Package Sensor_Definitions (continued)

   FOR Radar_Input USE
      RECORD
         Antenna_Azimuth AT 0 • Word Integer Range 0..13;
         First_Beacon_ID AT 1 • Word Integer Range 0..1;
         First_Beacon_Range AT 1 • Word Integer Range 2..13;
         Second_Beacon_ID AT 2 • Word Integer Range 0..1;
         Second_Beacon_Range AT 2 • Word Integer Range 2..13;
         Center_of_Scan_Sector AT 3 • Word Integer Range 0..13;
         In_Interrogate_Mode AT 4 • Word Integer Range 0..0;
         Search_Range AT 4 • Word Integer Range 1..1;
         Width_of_Scan_Sector AT 4 • Word Integer Range 2..3;
         Direction_of_Scan AT 4 • Word Integer Range 4..5;
         Rate_of_Scan AT 4 • Word Integer Range 6..7;
         Range_Profile AT 5 • Word Integer Range 0..199;
         Error_Flag AT 18 • Word Integer Range 0..15;
      END RECORD;

      Radar_Buffer : Radar_Input;
      Radar_Buffer_Address : Constant Integer
         := Radar_Buffer_Address;
      Radar_Input_Length : Constant Integer
         := 19;
   END Sensor_Definitions;
task body Sensor_Interface is
  use Sensor_Definitions;

  procedure Clear_the_DMA_and_the_Latch is ... end;
  procedure Set_up_the_DMA_for_the_next_burst is ... end;
  procedure Set_the_Latch_for_the_next_burst is ... end;

  pragma Priority(System'Max_Priority);

begin

loop

  accept DMA_finished_interrupt;
  Clear_the_DMA_and_the_Latch;
  Set_up_the_DMA_for_the_next_burst;

  select
    accept request_for_radar_input(output : out Sensor_Input)
    do output := radar_buffer;
    end;
  else
    send_error_message (radar_overrun);
    end select;

  Set_the_Latch_for_the_next_burst;

  end loop;

end Sensor_Interface;
PROCEDURE CLEAR_THE_DMA_AND_THE_LATCH IS
  USE LOW_LEVEL_IO;
BEGIN
  SEND_CONTROL ( DMA, ( CLEAR ) );
  SEND_CONTROL ( LATCH, ( CLEAR ) );
END CLEAR_THE_DMA_AND_THE_LATCH;

PROCEDURE SET_UP_THE_DMA_FOR_THE_NEXT_BURST IS
  USE LOW_LEVEL_IO;
BEGIN
  SEND_CONTROL ( DMA, ( SET_ADDRESS, RADAR_BUFFER_ADDRESS ) );
  SEND_CONTROL ( DMA, ( SET_COUNT, RADAR_INPUT_LENGTH ) );
  SEND_CONTROL ( DMA, ( SET_DIRECTION, INWARDS ) );
  SEND_CONTROL ( DMA, ( START ) );
END SET_UP_THE_DMA_FOR_THE_NEXT_BURST;

PROCEDURE SET_THE_LATCH_FOR_THE_NEXT_BURST IS
  USE LOW_LEVEL_IO;
BEGIN
  SEND_CONTROL ( LATCH, ( START ) );
END SET_THE_LATCH_FOR_THE_NEXT_BURST;
package Operator_Command_Definitions is

type Operator_Instruction is
  ( DMD_SKIN, DMD_SPLASH,
    HOME_CURSORS, PARK_CURSORS,
    AIM_RANGE_CURSORS, AIM_AZIMUTH_CURSORS,
    TOGGLE_AZIMUTH_OR_RANGE,
    ACKNOWLEDGE_ERROR,
    AUTO_ERASE, SLEW_WEAPON,
    RESTART, ARM, DISARM,
    UNIMPLEMENTED );

type Operator_Command ( instruction : Operator_Instruction ) is
  record
    case instruction is
    when AIM_CURSORS =>
      aim_direction : Screen_Direction;
      delta_index : Coordinate_Value;
    when others => null;
    end case;
  end record;

end Operator_Command_Definitions;
SEPARATE (Operator_Command_Handler)
Task body Operator_Input_Device.Reader is

  ...
PROCEDURE Convert_the_touch_to_a_command IS
  X, Y : Coordinate_Value;
  COMMAND_VECTOR : Integer Range 101 .. 1616;
BEGIN
  CASE COMMAND_VECTOR IS
    ...
    WHEN 1023 => COMMAND := (HOME_CURSOR);
    WHEN 1403 => COMMAND := (PARK_CURSOR);
    ...
    WHEN OTHERS => COMMAND := (UNIMPLEMENTED);
  END CASE;
END Convert_the_touch_to_a_command;

BEGIN  -- Operator_Input_Device.Reader
  LOOP
    READ_A_TOUCH;
    Convert_the_touch_to_a_command;
    CASE COMMAND_INSTRUCTION IS
      WHEN ARM | DISARM | UNIMPLEMENTED => NULL;
      -- ARM and DISARM are used by Convert_the_touch_  
      -- TO_A_COMMAND TO ARM OR DISARM THE TOUCH PANEL INPUT
      WHEN OTHERS => Send_next (COMMAND);
      -- REQUEST RENDEZVOUS WITH Operator_Command_Handler
      -- TO PASS A GOOD COMMAND TO IT
    END CASE;
  END LOOP;
END Operator_Input_Device.Reader;
task body Command_Dispatcher is
  use Command_Queue, Operator_Command_Definitions;
begin  -- Command_Dispatcher
  loop
    select
      accept Send_next ( Command : in Operator_Command );
      do Latest_command := Command;
      end Send_next;
      insert ( Latest_command );
    else
      select
        when (Current_command.instruction = Aim_range_cursor)
          or (Current_command.instruction = Aim_azimuth_cursor)
          or (Current_command.instruction = Home курсors)
          or (Current_command.instruction = Park курсors)
          or (Current_command.instruction = Toggle_azimuth_or_range)
          =>
            accept Acquire_next_cursor_operation
              (command : out Operator_Command )
            do Command := Current_command;
            end Acquire_next_cursor_operation;
        end select;
    end select;
  end loop;
end Command_Dispatcher;