REFERENCE MANUAL
FOR THE INTEL 432
EXTENSIONS TO ADA

Order Number: 172283-001
Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication, or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

- BXP
- CREDIT
- iR
- ICE
- iCS
- IMX
- Insite
- Intel
- int
- Intelelevision
- Intellec
- iOSP
- iRMX
- iSBC
- iSBX
- Library Manager
- MCS
- Megachassis
- Micromainframe
- Micromap
- Multibus
- Multimodule
- Plug-A-Bubble
- PROMPT
- RMX/80
- System 2000
- UPI

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

Ada is a trademark of the Department of Defense (Ada Joint Program Office).

VAX and VMS are trademarks of Digital Equipment Corporation.
<table>
<thead>
<tr>
<th>REV.</th>
<th>REVISION HISTORY</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-001</td>
<td>Original issue</td>
<td>12/81</td>
</tr>
</tbody>
</table>
Intel 432 Extensions to Ada

PREFACE

About this Manual


This manual is presented as part of the reference material for Intel's initial implementation of the Ada programming language. The initial implementation uses Intel's iAPX 432 Micromainframe products as execution vehicles for compiled programs. To fully accommodate the power of the iAPX 432, Intel Corporation has defined an extended Ada (hereafter referred to as 432 extended Ada). 432 extended Ada is a proper superset of Ada. Thus, any correct Ada program will compile and run correctly as a 432 extended Ada program.

The first compiler release is hosted by the VAX™ 11/750 or 11/780 system (under control of the VMS™ operating system) and generates code for the iAPX 432. This extensions manual does not discuss the initial compiler implementation and specifically does not describe unimplemented features of Ada in its first release. For details on the compiler and the unimplemented features, see the Intel 432 Cross Development System VAX/VMS Host User's Guide, Order Number 171870.

The Intel 432 Ada compiler is presently an incomplete implementation of the Ada programming language. It is intended that the Intel 432 Ada compiler will be further developed to enable implementation of the complete Ada programming language, and then be submitted to the Ada Joint Program Office for validation.

The descriptions in the following pages assume an understanding of the Ada language. Some code examples use a package called iMAX 432. This package is presumed to contain some services that might be made available by an iAPX 432 operating system. The details of the package are not specified herein and should not be considered important except for exposition purposes.

This manual is intended to introduce and define the extensions rather than to fully describe their use. Many of the code examples are non-compilable skeletons and are meant to serve merely as illustrations of correct usage.

The remainder of this manual is divided into two chapters as follows:

- Chapter 1 presents a general rationale for the design of the 432 extensions to Ada.
- Chapter 2 specifies the extensions, explains the motivation for each extension, gives some short code examples to illustrate usage, and lists the additions and/or revisions needed to incorporate each extension into the DoD Ada Manual.

* Ada is a trademark of the Department of Defense (Ada Joint Program Office).
Micromainframe is a trademark of Intel Corporation.
VAX and VMS are trademarks of Digital Equipment Corporation.
Related Intel Literature

The following list describes related Intel publications that are recommended as supplements to this book. Intel manual order numbers are listed and the address of the Intel Literature Department is on the back of the title page.


- **Intel 432 Cross Development System VAX/VMS Host User's Guide**. Order Number: 171870. This is a User's Guide for the first Intel implementation of an iAPX 432 extended Ada compiler. Programs are compiled on the VAX host. The resultant iAPX 432 code is downloaded to an iAPX 432 execution vehicle for execution and testing. This manual describes how to invoke and control the VAX-hosted compiler. Various other facilities for compiling and linking programs are also described.


The predecessor to this manual, *Engineering Specification for the iAPX 432 Extensions to Ada*, Order Number 171871, is obsolete.
# TABLE OF CONTENTS

1. DESIGN GOALS OF THE EXTENSIONS .......................................................... 1-1

2. THE INTEL 432 EXTENSIONS TO ADA ....................................................... 2-1
   2.1 432 Extended Ada, an Overview ......................................................... 2-1
   2.2 The ANY_ACCESS Type ......................................................................... 2-1
   2.3 Package Types ....................................................................................... 2-2
   2.4 Exporting Package Bodies ..................................................................... 2-5
   2.5 Ada Task Types vs. 432 Extended Ada Package Types ......................... 2-6
   2.6 Object Refinement and the ’at’ Operator ............................................. 2-6
      2.6.1 Example: Aliasing an Integer Variable .......................................... 2-8
      2.6.2 Example: Aliasing an Array Slice .................................................. 2-8
      2.6.3 Example: Refinements Used as Procedure Variables ..................... 2-8
      2.6.4 Example: Refinements to Dynamically Hide Attributes ................. 2-9
   2.7 Retyping Declarations ........................................................................... 2-10
      2.7.1 Example: Retyping a Variable Identifier ....................................... 2-10
      2.7.2 Restrictions on Use of ’retypes’ Declarations ................................. 2-11
1. DESIGN GOALS OF THE EXTENSIONS

A systems implementation language for Intel's iAPX 432 architecture should be a state-of-the-art, high-level programming language that conveniently exposes the concepts and capabilities of the iAPX 432.

Ada forms an ideal base for an iAPX 432 systems implementation language primarily because the design goals for Ada so closely match those of the iAPX 432. The main concepts and facilities of Ada are supported by the hardware facilities of the iAPX 432. However, Ada has been designed primarily to support the development of embedded systems, whereas the 432 also supports the development of dynamic systems. Embedded systems are characterized as static; new users, programs, and devices do not arise during user program execution. Thus, embedded systems have no need to deal with the spontaneous appearance of new entities and their new demands. Dynamic systems, on the other hand, are characterized by the appearance of new users, new programs, new devices, and new demands during program execution. Program execution in dynamic systems requires the ability to describe and manipulate entities defined and created after system initialization.

The following situations are typical of dynamic systems:

Implementation to be selected at execution time. A user wishes to define several alternative implementations of a package, desiring to programmatically select a specific implementation based on the execution-time needs of the system, e.g., a specific sorting algorithm is chosen based on the number of items to be sorted.

Implementation to be altered at execution time. A user wishes to suspend his program and replace the implementation of a package with a new one, e.g., terminal output is replaced with file output to limit information displayed to the console during a particular execution.

Implementation unknown. A user wishes to write programs that deal with other user programs or subprograms having unknown implementations, e.g., a program to graph functions is designed to accept arbitrary functions from other users.

Data structures partially unknown: A user wishes to supply a procedure that depends only on some aspects of the objects (i.e., data structure instances) it manipulates, allowing other aspects to remain unknown, e.g., a sort requires only that an integer be the key that it sorts, leaving unspecified any other parts of the objects to be sorted.

Data structures entirely unknown: A user wishes to write procedures that manipulate objects of arbitrary structure, either performing very general operations or investigating the object's type at execution time, e.g., garbage collection algorithms are required to manipulate arbitrary objects.

Ada supports applications such as those described above only if the user recompiles those parts of his system that were unknown or changing and then restarts his program. The iAPX 432 architecture supports these dynamic applications directly. However, in order to use these features, a systems implementation language must effectively describe such manipulations without requiring recompilation. The 432 extensions to Ada enable effective description of dynamic manipulations, allowing both compile-time and execution-time type-checking.
2. THE INTEL 432 EXTENSIONS TO ADA

2.1. 432 Extended Ada, an Overview
Intel has defined and currently supports four constructs as extensions to Ada:

1. the predefined type: ANY-ACCESS,
2. the package typing phrase: package type,
3. the refinement operator: at, and
4. the keyword: retypes.

These extensions are easily grasped by users who know Ada. Furthermore, no "unlearning" is required since Ada is a proper subset of 432 extended Ada.

All the extensions are aimed at increasing the power of the language in dealing with dynamically defined entities. They allow users to manipulate entities whose definitions were compiled after some parts of the user program began execution. They allow users to manipulate entities whose implementations may change dynamically.

The following sections detail each extension individually.

2.2. The ANY_ACCESS Type
The 432 extensions supply a predefined type called ANY_ACCESS. Variables of type ANY_ACCESS may be assigned values of any Ada access type (i.e., any identifier typed as access), provided that the access values are explicitly qualified to be of type ANY_ACCESS. Variables of type ANY_ACCESS cannot be dereferenced directly; instead, they must be qualified to be of a specific access type via UNCHECKED_CONVERSION. The ANY_ACCESS type provides a simple way for programs to manipulate entities whose types are unknown until execution time, as often occurs in operating systems and utility programs. For example, the following procedure writes out a (access) data structure as a string of N bytes onto an external file:

```ada
procedure DUMP( ANY_USER_DATA_OBJECT: ANY_ACCESS) is
  type MEMORY_IMAGE is array( SHORT_ORDINAL range 0 .. 65535 ) of BYTE;
  -- assumes maximum size
  type OBJECT_TYPE is access MEMORY_IMAGE;
  -- any operations such as
  --   ANY_USER_DATA_OBJECT.WHATEVER
  -- are always illegal, since no structure is assumed for
  -- objects reached through ANY_ACCESS values. Unsafe
  -- conversions are used to assert structure:
  function CONVERT_TO_OBJECT is new
    UNCHECKED_CONVERSION( ANY_ACCESS, OBJECT_TYPE );
  OBJECT: OBJECT_TYPE := CONVERT_TO_OBJECT( ANY_USER_DATA_OBJECT );
  N: SHORT_ORDINAL :=
    iMAX432.LENGTH_OF_SEGMENT ( ANY_USER_DATA_OBJECT);
  -- this iMAX432 function returns actual length - 1
begin
  for i in 0 .. N loop
    iMAX432.WRITE_BYTE( OBJECT( i ) );
  end loop;
end DUMP;
```

2-1
Intel 432 Extensions to Ada

A user might store a byte image of any data segment by:

\[
\text{DUMP( ANY_ACCESS(SOME_OBJECT) );}
\]

-- where SOME_OBJECT is a value of some access type.

Changes to DoD Ada Manual for ANY_ACCESS:

- Incorporate Section 2.2.2 above as new Section 3.8b.
- Add "type ANY_ACCESS is access implementation_defined;" to package STANDARD in Appendix C.

2.3. Package Types

Although Ada supports the dynamic creation of non-library level packages, it does not support the dynamic creation of library level packages. On the iAPX 432, packages are supported as domain objects, and therefore behave as values. The 432 extensions to Ada also support dynamic packages by allowing package values and therefore package types. These package types behave similarly to Ada record types, with package bodies performing as record aggregates in creating values of package types.

Package types are provided by allowing the keyword \texttt{type} to follow \texttt{package} in package specification headings. For example:

\[
\begin{align*}
\textit{type} & \quad \text{ITEM-TYPE is} \\
& \text{record} \ldots \\
& \text{end record;} \\
\text{package type} & \quad \text{LIST-PACKAGE is} \\
& \quad \text{type} \ \text{LIST-TYPE is private;} \\
& \quad \text{function FIRST (L : LIST-TYPE) return ITEM-TYPE;} \\
& \quad \text{EMPTY-LIST : exception;} \\
& \quad \text{procedure ADD( ITEM : ITEM-TYPE; LIST : LIST-TYPE );} \\
& \ldots \\
& \text{private} \\
& \quad \text{type} \ \text{LIST-TYPE is new ANY_ACCESS;} \\
& \ldots \\
& \text{end} \ \text{LIST-PACKAGE;} \\
\end{align*}
\]

The type \texttt{LIST-PACKAGE} describes an abstraction that manipulates lists whose elements are of type \texttt{ITEM-TYPE}. No particular implementation is assumed for the package \texttt{LIST-PACKAGE}; indeed, several implementations may coexist or appear dynamically.

A specific instance of a package type (i.e., a package type value) is created by declaring a constant of the package type:

\[
\begin{align*}
\text{package} & \quad \text{LIST-AS-ARRAY is constant} \ \text{LIST-PACKAGE;} \\
\end{align*}
\]

The 432 extended Ada compiler interprets this declaration as an announcement that a specific package is to be created. The declaration of the constant \texttt{LIST-AS-ARRAY} given above is equivalent to an Ada package specification repeating the declarations associated with \texttt{LIST-PACKAGE}. Thus, the declaration above is equivalent to the declarations:
type LIST_TYPE is private;
function FIRST (L : LIST_TYPE) return ITEM_TYPE;
EMPTY_LIST : exception;
procedure ADD (ITEM: ITEM_TYPE; LIST: LIST_TYPE);
...
private
type LIST_TYPE is new ANY_ACCESS;
...
end LIST_AS_ARRAY;

A package body is expected to accompany every package specification, so the compiler expects a body for LIST_AS_ARRAY. In the following package body, the LIST_PACKAGE is implemented using an array representation for lists:

package body LIST_AS_ARRAY is
type LIST_REP_ARRAY is
  record
    NUMBER_IN_LIST : INTEGER range 0 .. 100 := 0;
    VALUE : array( 1 .. 100 ) of ITEM_TYPE;
  end record;
type ARRAY_LIST is access LIST_REP_ARRAY;
function CONVERT is new UNCHECKED_CONVERSION (LIST_TYPE, ARRAY_LIST);
function FIRST (L : LIST_TYPE) return ITEM_TYPE is
  LIST : ARRAY_LIST := CONVERT(L); -- unchecked type conversion
begin
  if LIST.NUMBER_IN_LIST = 0 then
    raise EMPTY_LIST;
  else
    return LIST.VALUE( 1 );
  end if;
end FIRST;
...
end LIST_AS_ARRAY;

Instances of package types are often defined (again, by declaring a constant of the package type) in scopes other than the scope of the package type declaration itself. The lifetime of such instances is determined by the scope of the package type declaration and not the scope of the particular package constant. Such package instances may not refer to entities defined in scopes with a shorter lifetime than the scope of the package type declaration (except for initialization code within the package declaration itself).

Once created, the package constant LIST_AS_ARRAY behaves as a normal Ada package. Programmers can access its public attributes using the dot notation (e.g., "LIST_AS_ARRAY.FIRST( X )") and can open the package with a use clause.

The full power of dynamic packages is obtained with package variables. Programmers can declare variables (or record fields, etc.) of package types:

A_LIST : LIST_PACKAGE;

Variables of package types have public attributes that can be accessed either by dot notation or use. However, no knowledge of the implementation of the package can be assumed. Indeed, the implementation may change dynamically.

As an example, lists can also be implemented as linked structures:
package LIST_AS_LINKS is constant LIST_PACKAGE;

package body LIST_AS_LINKS is

  type LIST_REP_LINKS is
  record
    ITEM : ITEM_TYPE;
    NEXT : LINKED_LIST;
  end record;

function CONVERT is new UNCHECKED_CONVERSION (ITEM_TYPE, LIST_LINKED);

function FIRST (L : LIST_TYPE) return ITEM_TYPE is

  LIST : LINKED_LIST := CONVERT(L); -- unchecked type conversion

begin
  if LIST = null then
    raise EMPTY_LIST;
  else
    return LIST.ITEM;
  end if;

end FIRST;

end LIST_AS_LINKS;

Users of the package type LIST_PACKAGE can decide at execution time on the implementation they prefer:

LIST_HANDLER : LIST_PACKAGE;
MY_LIST : LIST_HANDLER.LIST_TYPE;
MY_ITEM : ITEM_TYPE;

if NUMBER_ITEMS_EXPECTED <= 100 then
  LIST_HANDLER := LIST_AS_ARRAY;
else
  LIST_HANDLER := LIST_AS_LINKS;
end if;

MY_ITEM := LIST_HANDLER.FIRST( MY_LIST );

WARNING: Use of ANY_ACCESS prevents any type checking on the type of the list. If a programmer should somehow create a LIST_AS_ARRAY list and inadvertently pass that list to a LIST_AS_LINKS operation, no exception will be raised.

A common use of dynamic package implementation is input/output. Users will normally write their programs so as to perform I/O to a file package value. The file can be implemented as a disk file, a temporary file, a terminal or a line-printer depending on decisions made during the execution and debugging of the program.

Another important use of package types is in providing subprogram variable facilities. The following package type can be employed in a plotting/graphing package to describe the function being graphed:

package type FUNCTION_TO_DISPLAY is
  function F( X : REAL ) return REAL;

24
An operating system for the Intel 432 might use package types to define all its entities, even at the physical I/O level. Use of package types allows such an operating system to reconfigure itself as physical devices come and go, and as user programs appear and disappear.

Changes to the DoD Ada Manual for package types:
- Incorporate Section 2.3 above as new Section 7.3a.
- Mention the existence of package types in Section 3.3.
- Modify the BNF grammar as follows:

- Package variables can, in general, be used wherever Ada packages can be used, except in combination with the generic keyword. The semantics of package variables and Ada packages are the same, except that for package variables, the specific package implementation, i.e., the associated package body, may not be known until (or may change during) program execution.

2.4. Exporting Package Bodies
When a package constant (that is, instance) is assigned to a package variable in an outer scope, the package body may be exported outside the scope in which it was defined. This exporting is legal only so long as the package instance is independent of the intervening scopes. If use of an exported package results in an attempt to access information from nonexistent scopes, the hardware of the iAPX 432 will raise the exception INVALID_OBJECT_ACCESS.

For example:

\[ X : LIST_PACKAGE ; \]
Intel 432 Extensions to Ada

procedure DISAPPEARING is
  ILLEGAL_VAR: INTEGER := 0;
  L: constant LIST_PACKAGE;
package body L is
...;
  LEGAL: INTEGER := ILLEGAL_VAR + 1; -- legal, since
      -- ILLEGAL_VAR exists at elaboration.
function FIRST( L: LIST_TYPE ) return ITEM_TYPE is
begin
  ...;
  ILLEGAL_VAR := ILLEGAL_VAR + 1; -- illegal, since
      -- ILLEGAL_VAR will not exist when return is made from
      -- DISAPPEARING, yet X still allows calls to L.FIRST.
end FIRST;
...;
end L;

begin
  X := L; -- exports package constant L
end DISAPPEARING;

MY_LIST : X.LIST_TYPE;
MY_ITEM : ITEM_TYPE;
...
DISAPPEARING;
MY_ITEM := X.FIRST( MY_LIST ); -- causes INVALIDOBJECTACCESS
      -- exception, since the instantiation of DISAPPEARING and
      -- therefore the variable ILLEGAL_VAR no longer exist.

2.5. Ada Task Types vs. 432 Extended Ada Package Types
Ada task types have properties different from the 432 extended Ada package types.
The 432 package types behave similar to Ada record types, with package bodies performing as record aggregates in creating instances.
However, Ada task types are limited private. They can be used only in variable and parameter declarations. Further, declaring a variable or field of a given task type has the side effect that (at elaboration) an instance of the task is created. Finally, Ada task types permit only one implementation to exist for a given compilation, since a single task body accompanies each task type.

2.6. Object Refinement and the 'at' Operator
The object refinement facility of 432 extended Ada enables programmers to create aliases to existing objects or components within objects. Object refinement is directly supported in the hardware of the iAPX 432 and therefore all powers and limitations of the hardware are visible in the extended language. A refinement can be specified on arrays, records, packages and single data objects.
A refinement is created by execution of the refinement operator `at`. The general form for the syntax is as follows:

```plaintext
type_mark at variable_name
```

*Type_mark* must be an access type providing access to a type whose structure matches the structure of the contiguous set of attributes beginning at *variable_name*.

A refinement of a single data object must specify the object being refined. The *type_mark* must be an access to the type of the object being refined.

One-dimensional arrays can be refined by specifying the array element at which the refinement begins. The *type_mark* must be an access to an array subtype having the same base type as the array being refined.

Records and packages can be refined by specifying the name of the first *component* in the refinement. When the refinement components are all subprograms, the type of the refinement must be a package type. When the refinement comprises two or more data objects, the type of the refinement must be a record type. Refinement is not possible for a mixture of subprograms and entities of type other than *access*. Structural equivalence is used in matching the contiguous elements in record and package refinements.

Several general rules govern the refinement facility:

1. For two structures to match, the attributes must pair up position-wise, with both attributes in the pair having the same type. If any attribute is a constant, its matching attribute must be a constant with an equal value. If any attribute is a subprogram or package, its matching attribute must be a subprogram with matching parameter structure or a package with matching visible part and private part structure.

2. Refinement is restricted to contiguous sections of existing objects. When more than one refinement from a given object is desired, the layouts of the object and its refinements must be chosen carefully. In some cases, no set of layouts will enable all of the desired refinements to be made.

3. No refinement may involve the variant part of a record unless that variant is a subrecord.

4. All refinements must begin on a byte boundary.

5. The addressing structure of each attribute in a refinement must match that expected by the refinement. The legality of refinements can therefore depend on the storage layout algorithms of the 432 extended Ada compiler. The compiler defines six attribute classes:
   1. Access values, including user access-type, package, and task variables, and elaboration-time access constants;
   2. User access-type compile-time constants;
   3. Other access constants, including subprograms, packages, tasks;
   4. Data values, including user data variables and elaboration-time data constants;
   5. Data constants, including compile-time data constants;
   6. Others, including types (not TYPE_DESCRIPTION variables);

As a rule, the compiler allocates storage such that refinements may never include attributes of more than one of these classes, and may never include attributes of class 5.
Intel 432 Extensions to Ada

(6) An object continues to exist as long as any refinement to it exists.
The value returned by the at operator is an access value of the specified type which
provides access to the specified set of attributes.

2.6.1. Example: Aliasing an Integer Variable

type INTEGER_ACCESS is access INTEGER;
INT : INTEGER := 432;
INT_ACCESS : INTEGER_ACCESS := INTEGER_ACCESS at INT;
-- INT and INT_ACCESS all both refer
-- to the same object, whose value is 432

2.6.2. Example: Aliasing an Array Slice

type DOSSIER is
  record
    NAME : STRING(1..21);
    ADDRESS : STRING(1..50);
    PHONE : STRING(1..10);
  end record;

type DAILY_QUOTA_ARRAY is array (1..10) of DOSSIER;
type DAILY_QUOTA is access DAILY_QUOTA_ARRAY;
MAILING_LIST : array (1..500) of DOSSIER := ( ... );
WORKER ASSIGNMENT : DAILY_QUOTA := DAILY_QUOTA at MAILING_LIST(108);

A refinement that specifies a package type along with a matching set of contiguous
attributes for some existing object returns an access value that provides access to a
refined package having those attributes.

When a refinement specifies a package type which consists entirely of one subprogram
and the single designated variable is a matching subprogram, then the returned value
provides access to that designated subprogram. This access value provides a handle
for the subprogram that may be transferred among access variables of the same pack­
age type (eg, passed as a parameter).

2.6.3. Example: Refinements Used as Procedure Variables

Consider the following two packages, INTEGRATION_ROUTINES, a library package, and
FUNCTIONS_OF_INTEREST, containing user-defined functions:

package INTEGRATION_ROUTINES is
  -- assume this package exists inside a
  -- library package MATH_LIB

package type INTEGRAND is
  function F(X:REAL) return REAL;
end INTEGRAND;

function ROMBERG_RULE( FX:INTEGRAND; START,STOP:REAL )
  return REAL is
begin ... end ROMBERG_RULE;
function SIMPSON_RULE (FX: INTEGRAND; START, STOP: REAL) return REAL is begin ... end SIMPSON_RULE;
...
end INTEGRATION_ROUTINES;

package FUNCTIONS_OF_INTEREST is
  function F1(X: REAL) return REAL is begin return X**2; end F1;
...
end FUNCTIONS_OF_INTEREST;

Use of the at operator to select a refinement of the library routines is achieved by the following:

procedure MY_CALCS is
  VALUE : REAL;
begin
  ...
  VALUE := ROMBERG_RULE(INTEGRAND at FUNCTIONS_OF_INTEREST.F1,
    START:=0.0, STOP:=1.0);
  ...
end MY_CALCS;

2.6.4. Example: Refinements to Dynamically Hide Attributes

Refinements may also be used to dynamically hide attributes. A user may define a package and then hand out refinements of that package to various users, hiding various attributes from different users:

type ITEM_TYPE is ...;
type DB is ...;
type DB_REPRESENTATION is access DB;
DB : DB_REPRESENTATION := new DB ( ... );

package type DATA_ENTRY is
  procedure ADD(ITEM: ITEM_TYPE);
end DATA_ENTRY;

package type READ_ONLY is
  function ASK(QUERY_ITEM: ITEM_TYPE) return BOOLEAN;
end READ_ONLY;

package type CORRECTION is
  function QUERY(ITEM: ITEM_TYPE) return BOOLEAN;
  procedure MODIFY(BAD_ITEM: ITEM_TYPE);
end CORRECTION;
Intel 432 Extensions to Ada

```ada
package DB_MGR is
  DATA_BASE : constant DB_REPRESENTATION := DB;
procedure INSERT(NEW_ITEM : ITEM_TYPE);
function QUERY(ITEM : ITEM_TYPE) return BOOLEAN;
procedure UPDATE(UPDATED_ITEM : ITEM_TYPE);
end DB_MGR;

package body DB_MGR is 
end DB_MGR;
```

With the above specifications, the following refinements are possible:

- DATA_ENTRY at DB_MGR. INSERT
- READ_ONLY at DB_MGR. QUERY
- CORRECTION at DB_MGR. QUERY

Changes to the DoD Ada Manual for Object Refinement:
- Add a sixth production for "expression" in Section 4.4 and in Appendix E: "expression ::= name at name".
- Incorporate Section 2.6 above as new Section 4.7a.

2.7. Retyping Declarations

Ada allows the conversion of a value from one type to another. The Intel 432 extensions to Ada allow the conversion of an object to a new type. Specifically, the keyword `retypes` is permitted in place of the Ada keyword `renames` for object declarations.

```
retyping_declaration ::= type_mark retypes name
```

Use of `retypes` suspends type-checks so that the new name and the old name may have different types.

2.7.1. Example: Retyping a Variable Identifier

```
Z: INTEGER;
type TWO_HALF_WORDS is
  record
    H1, H2: SHORT_ORDINAL;
    -- SHORT_ORDINAL is a 432 predefined type.
  end record;
X: TWO_HALF_WORDS retypes Z;
```

Given these declarations, X.H2 now references the high-order 16 bits of Z. That is, adding one to X.H2 has the effect of adding 65536 to Z except in the case of overflow.

The `retypes` facility should not be used where use of UNCHECKED_CONVERSION is possible. The `retypes` facility is more dangerous since it preserves aliases of different types for the same object, while UNCHECKED_CONVERSION returns a copy of a value of one type as a new value of another type. In general, `retypes` should be used only when a specific location in an object must be examined as different types.
2.7.2. Restrictions on Use of 'retype' Declarations

Two restrictions exist on the use of retyping declarations:

(1) values of type access may not be converted to values of a type other than access, and
(2) values of a type other than access may not be converted to values of type access.

Changes to the DoD Ada Manual for Retyping Declarations:

- Incorporate Section 2.7 above as new Section 8.5a.
- Add a new production for "declaration" in Section 3.1 and Appendix E: "declaration ::= retyping_declaration".
- Add a new production in Appendix E: "retyping_declaratioon ::= type_mark retypes name".
- Add the keyword "retype" to the reserved word list in Section 2.9.
REQUEST FOR READER’S COMMENTS

Intel’s Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating).

NAME ___________________________ DATE ______________
TITLE __________________________ STATE _____________ ZIP CODE ____________
COMPANY NAME/DEPARTMENT ___________________________
ADDRESS __________________________________________
CITY __________________________ STATE ______________ ZIP CODE ____________
(COUNTRY)

Please check here if you require a written reply. ☐
WE'D LIKE YOUR COMMENTS...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 79 BEAVERTON, OR
POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
SSO Technical Publications, WW1-487
3585 SW 198th Ave.
Aloha, OR 97007

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES