intel®

# iMAX 432 Reference Manual

172103-002

# iMAX 432 REFERENCE MANUAL

Order Number: 172103-002

| REV. | REVISION HISTORY | DATE |
|---|---|---|
| -001 | Original issue | 12/82 |
| -002 | Revised for iMAX V2 | 05/82 |

# PREFACE

This manual describes iMAX 432, Intel's Multifunction Applications Executive for the Intel 432 Micromainframe computer system. iMAX enhances the 432's unique architectural support for storage management, concurrent processing, and other operating system functions. iMAX provides a simple and uniform functional interface to the user.

## iMAX USERS

iMAX is designed to provide executive services to user-supplied software that calls on iMAX. iMAX also provides a run-time environment for Ada and other high-level languages. iMAX does not currently provide a command language or a human interface. iMAX users are not terminal operators; instead they are system designers and programmers who design, implement, and validate software that calls on iMAX.

## iMAX CONFIGURATIONS

iMAX is a configurable operating system. Users can choose which subsystems (packages) to incorporate in their applications. Intel supplies two iMAX configurations:

| | |
|---|---|
| full iMAX | provides the run-time support needed by the Ada* language, including dynamic storage management, dynamic process management, and an extensible I/O interface. |
| minimal iMAX | is a subset of full iMAX that provides a simple environment for executing multiple processes on multiple 432 processors. Minimal iMAX does not provide dynamic storage management or dynamic process management. Minimal iMAX does not provide I/O except through the DEBUG-432 debugger. Minimal iMAX cannot support full Ada. The advantages of minimal iMAX are its small memory requirement, its similarity to iMAX Version 1 (for users transporting iMAX V1 programs), and its simple run-time environment. |

This manual concentrates on describing full iMAX. Information specific to minimal iMAX is contained in an appendix.

iMAX also supports separate configuration of Peripheral Subsystems for input/output. Users can add their own device drivers and extend the I/O configuration.

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

## iMAX RELEASES

iMAX Version 2 (V2) supercedes iMAX V1. However, the minimal configuration of V2 provides almost exactly the same interface to user programs as iMAX V1. Users of iMAX V1 can easily recompile and relink their programs to use the V2 minimal configuration -- the Ada Compiler System will detect any mismatch in interfaces caused by the change.

Version 3 of iMAX will adapt iMAX to Release 3 of the iAPX 432 architecture. There will be significant changes in the iAPX 432 architecture, many of them visible to iMAX users. iMAX V3 will also extend iMAX functionality to support virtual memory.

**CAUTION**

Many user interfaces to iMAX may change in the future because of either Release 3 architecture changes, iMAX reorganization, or both. For this reason, users of iMAX should encapsulate their use of iMAX within interface packages that provide services to applications software while centralizing dependence on the iMAX interface. User code should not rely on any particular physical layout of system objects or user-defined objects, because object representations will be affected by Release 3 architecture changes.

## PROGRAMMING LANGUAGES

iMAX is designed to be independent of any particular programming language. Initially, Ada is the only programming language provided for the 432, and the iMAX interface described in this manual is a set of Ada package specifications, each corresponding to a particular iMAX service. This interface is uniform, modular, and uses the Ada compilation and binding system to configure iMAX and to type-check all calls to iMAX. iMAX is itself written in Ada and benefits from Ada's support of reliable modular programming.

The Intel 432 Ada compiler is presently an incomplete implementation of the Ada progamming 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.

Also supplied as part of iMAX is software to control Peripheral Subsystems (PSs) used for initialization and input/output. The PS software is written in Intel's PL/M-86 programming language. The PS software executes on an Intel 8086 Attached Processor (AP) supported by Intel's iRMX 88 Real-Time Multitasking Executive.

## ADA EXCEPTIONS

**CAUTION**

Both iMAX and the examples in this manual are designed to use the Ada exception-handling facility. However, the Intel 432 Ada compiler does not yet support Ada exceptions.

Wherever this manual describes iMAX raising an exception, iMAX actually calls an internal iMAX procedure that causes the process in which the exception occurs to fault. How the resulting fault is handled is described in Appendix FLT, Fault-Handling.

Wherever the examples in this manual raise an exception, the Intel 432 Ada compiler generates no operation (raise statements are ignored).

## iMAX LISTINGS

The listings of iMAX modules in this manual have been modified in several ways. For Ada packages, the environment pragmas and private parts have been removed. All listings have been reformatted and paginated. Where listings are known to contain erroneous comments, corrections have been inserted, flagged by "!!". The iMAX user also receives files containing the actual source listings, as specified in Appendix SUM, Software Components Summary.

## EXAMPLES

Major examples in this manual (those at the end of chapters KEY, DEF, BPM, COM, PEN, and EXT) have been successfully compiled. Minor examples (those that appear in the text of chapters) have not been compiled. None of the examples have been tested. Examples should not be considered definitive or ideal implementations of their functions.

## PREREQUISITES

To use this manual, you should understand the basic concepts of the 432 architecture and the Ada programming language. Specifically, you should understand:

- the 432's object addressing and protection mechanism

- Ada access types and values, Ada data typing, the Ada exception facility, and the organization of an Ada program into packages

## REFERENCES

Introductory information on the iAPX 432 architecture is provided by:

Intel 432 System Summary:
Manager's Perspective
Order Number: 171867

Introduction to the iAPX 432 Architecture
Order Number: 171821

iAPX 432 Object Primer
Order number: 171858

The Ada programming language is described in:

Reference Manual for the Ada Programming Language
Order Number: 171869

Intel's extensions to Ada are described in:

Reference Manual for the
Intel 432 Extensions to Ada
Order Number: 172283

Users of some iMAX packages require more information about the iAPX 432 architecture than is in this manual. The authoritative references for the iAPX 432 architecture are:

iAPX 432 General Data Processor
Architecture Reference Manual
Order Number: 171860

iAPX 432 Interface Processor
Architecture Reference Manual
Order Number: 171863

iMAX applications are developed using an Intel 432 Cross Development System. These manuals describe the 432 CDS:

Introduction to the Intel 432 Cross Development System
Order Number: 171954

Intel 432 Cross Development System
VAX Host User's Guide
Order Number: 171870

Intel 432 Cross Development System
Workstation User's Guide
Order Number: 172097

Intel 432 Cross Development System
Ada Support Packages User's Guide
Order Number: 172521

Mainframe Link for Distributed Development
User's Guide
Order Number: 121565

Asynchronous Communication Link User's Guide
Order Number: 172174

iMAX can run on 432/600 systems, including the System 432/670 execution vehicle of the Intel 432 Cross Development System. 432/600 systems are described in:

System 432/600 System Reference Manual
Order Number: 172098

System 432/600 Diagnostic Software User's Guide
Order Number: 172099

The portion of iMAX that runs on the 8086 Attached Processor (AP) is written in Intel's PL/M-86 programming language and uses Intel's iRMX 88 Real-Time Multitasking Executive. Configuration of iMAX and user AP software requires Intel's iRMX 80/88 Interactive Configuration Utility. These products are described in:

PL/M-86 User's Guide
Order Number: 121636

PL/M-86 Programming Manual
for 8080/8085-Based Development Systems
Order Number: 9800466

[1]VAX is a trademark of Digital Equipment Corporation.

<u>PL/M-86 Compiler Operating Instructions</u>
<u>for 8080/8085-Based Development Systems</u>
Order Number: 9800478

<u>Introduction to the iRMX 80/88 Real-Time</u>
<u>Multitasking Executives</u>
Order Number: 143238

<u>iRMX 88 Reference Manual</u>
Order Number: 143232

<u>iRMX 88 Installation Instructions</u>
Order Number: 143241

<u>iRMX 80/88 Interactive Configuration Utility User's Guide</u>
Order Number: 142603

## MANUAL ORGANIZATION

iMAX is a modular system and is described by a modular manual. Sections will be added to this manual as packages (modules) are added to iMAX. Users can remove chapters or appendices that do not apply to the iMAX configuration they are using. For example, a user of the minimal iMAX configuration does not need Chapter BPM, Basic Process Management. To make changing the manual easier, chapters and appendices have mnemonic identifiers instead of being numbered or lettered in sequence (e.g., Chapter KEY instead of Chapter 1).

This manual is organized to present the most generally useful material first, with implementation information in the later part of the manual. The appendices present configuration-specific and version-specific reference information needed by system designers and implementers.

Some iMAX facilities are usually accessed through more convenient higher-level facilities available in Ada or the 432 extensions to Ada, including most of the facilities described in these chapters:

| | |
|---|---|
| STO | Storage Management |
| PEN | Program Environment Access |
| IO | Input/Output |

These facilities are provided for the use of systems programmers implementing language run-time environments, database systems, or other systems software.

Users should read Chapters KEY (Key Concepts), HOW (How to Use iMAX), and DEF (Basic Definitions) first. The other chapters can be read in any order.

This manual is organized as follows:

- Chapter KEY, Key Concepts -- the distinctive philosophy and organization of iMAX as a collection of Ada packages that cooperates with and complements the hardware architecture.

- Chapter HOW, How to Use iMAX -- an introduction to how Ada programmers reference iMAX services and convert data to and from the types used by iMAX.

- Chapter DEF, Basic Definitions -- an Ada interface to the access descriptors, object descriptors, rights, and types defined by the hardware architecture.

- Chapter STO, Storage Management -- how iMAX uses storage resource objects to support the creation and reclamation of objects.

- Chapter BPM, Basic Process Management -- how iMAX extends the hardware-defined operations on processes to create a tree structure for processes and to let processes control the operation of other processes.

- Chapter COM, Interprocess Communication -- an Ada interface to the hardware-defined port operations for transferring messages between processes.

- Chapter PEN, Program Environment Access -- an Ada interface to the 432 system objects corresponding to the access environment of a module activation.

- Chapter EXT, Extended Types -- user-defined types and the operations involving them.

- Chapter IO, Input/Output -- I/O as viewed by user programs running on 432 GDPs.

- Chapter IOI, Input/Output Implementation -- the iMAX software that runs on the Attached Processor (AP) and that controls the Interface Processor (IP); how to add a device driver to iMAX.

- Chapter CON, Configuration -- how to specify to iMAX the processors and processes in a system at initialization.

- Chapter INI, Initialization -- how an iMAX system initializes; how to load and start an iMAX system from AP software or from the DEBUG-432 debugger.

- Appendix SUM, Software Components Summary -- the files that make up the iMAX 432 product received by customers, and their correspondence to listings and descriptions in this manual.

- Appendix MIN, Minimal Configuration -- the packages and limitations of the minimal configuration of iMAX, and how the minimal configuration differs from the full configuration.

- Appendix RGT, User-Visible Type Rights -- type rights for system object types that can be referenced by user programs.
- Appendix FLT, Fault-Handling -- iMAX action when a fault occurs, as seen by user software.
- Appendix HDW, Hardware Configuration Information -- what assumptions iMAX makes about 432 components and about jumper settings in the Peripheral Subsystem.
- Appendix PRS, Process Scheduling Information -- scheduling information for iMAX system processes that compete with user processes to be dispatched, and guidelines for selecting user process scheduling parameters.
- Appendix SIZ, Size Information -- bytes of memory required by iMAX, and by each added instance of selected system objects.
- Appendix PRF, Performance Information -- execution times for selected iMAX operations.
- GLO, Glossary -- important terms used in this manual.

## NOTATION

These notational conventions are used in Ada specifications and examples:

- Package names have the first letter of each word in upper case and other letters in lower case (unless the name includes an acronym, e.g., package "iMAX_Definitions"). Subprogram names have the first letter of the first word in upper case, e.g., "Check_object type". Other identifiers, even if predefined by Ada or by the 432 extensions to Ada, appear in all lower case (unless they are acronyms), e.g., "constraint_error".
- Each package or subprogram specification is documented with a Function paragraph that explains what it does, but not how it is implemented.
- Any explicitly-raised exceptions are listed in an Exceptions paragraph.

Ada identifiers used in the narrative appear as they do in program text and are underscored when introduced (e.g., new identifier).

## ABBREVIATIONS

These abbreviations are used in this manual:

| | |
|---|---|
| AD | Access Descriptor |
| AP | Attached Processor |
| EAS | Entered Access Segment |
| GDP | iAPX 432 General Data Processor |
| IP | iAPX 432 Interface Processor |
| IPC | Interprocessor Communication |
| OD | Object Descriptor |
| OT | Object Table |
| PS | Peripheral Subsystem |
| PSO | Physical Storage Object |
| RCO | Refinement Control Object |
| SRO | Storage Resource Object |
| TCO | Type Control Object |
| TDO | Type Definition Object |

## ILLUSTRATIONS

## TABLES

# CHAPTER KEY
# KEY CONCEPTS

This chapter reviews basic concepts needed to understand iMAX:

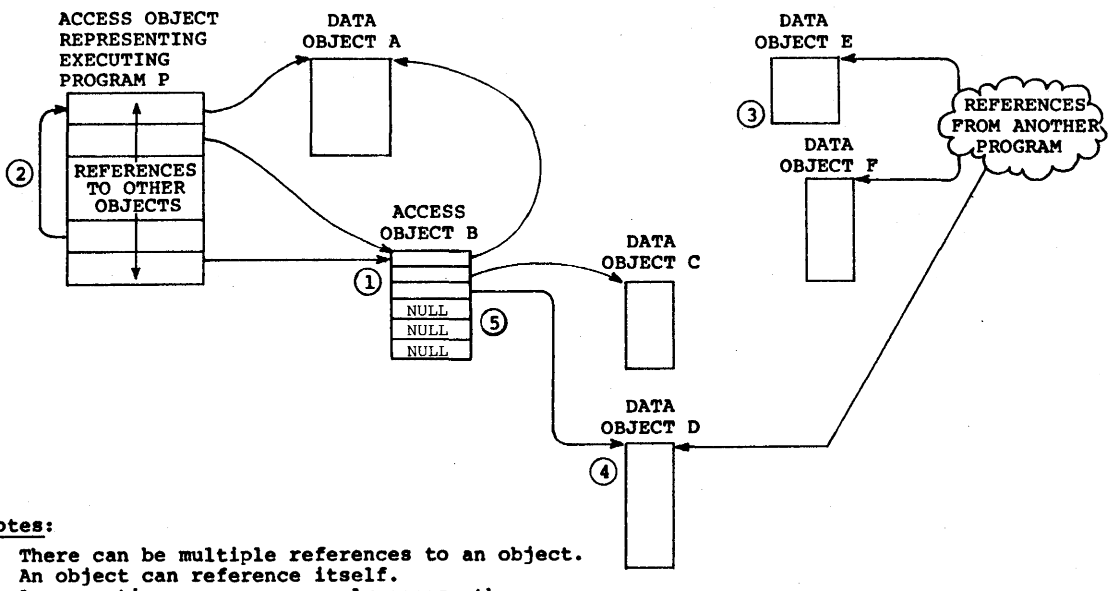
- the organization of a 432 system as a network of objects
- the inclusion of operating system functions in the 432 hardware architecture
- the system objects used to implement iMAX
- modular programming using Ada packages

The chapter ends with an example of modular programming using iMAX and Ada.

## OBJECTS

All data and program structures in a 432 system are contained in objects. Memory in a 432 system is addressed as a network of objects, not as an array of bytes or words. Each basic object can contain either data or references to other objects. A cluster of related objects tied together by references can be treated as a single object, allowing hierarchies of objects to be constructed. An executing program is itself represented by an object, and can only access objects for which it has references. This structured memory organization embodies the logical structure of programs (as networks of subprograms, arrays, pointers, and other abstractions) in the run-time structure of memory.

References to objects are called access descriptors (ADs) and are contained in access objects. Any other kind of information in a 432 system is contained in data objects. Data objects cannot contain ADs; access objects can contain nothing but ADs. The 432 architecture carefully controls operations on ADs to ensure that ADs cannot be forged or corrupted in any way. An access descriptor contains an index for the referenced object, and also access rights bits that grant or deny permission to execute particular operations on the object using the AD. Different ADs for an object can grant different rights. For example, two programs can both have ADs for a shared data object, with one of the programs having only read access and the other program having only write access. An AD can also have the value null, which references no object. For Ada programs, each Ada access value corresponds to an AD, and the term "access descriptor" derives from this correspondence.


ACCESS OBJECT REPRESENTING EXECUTING PROGRAM P
DATA OBJECT A
DATA OBJECT E
REFERENCES FROM ANOTHER PROGRAM
DATA OBJECT F
2
REFERENCES TO OTHER OBJECTS
ACCESS OBJECT B
DATA OBJECT C
1
NULL
NULL
NULL
5
3
DATA OBJECT D
4

**Notes:**

1. There can be multiple references to an object.
2. An object can reference itself.
3. An executing program can only access the objects for which it has references. For example, Program P cannot access Data Objects E or F.
4. Different programs in a 432 system can reference shared objects.
5. A reference can be null and designate no object.

F-0289

Figure KEY-1. How 432 Memory is Structured as a Network of Objects

Because there can be multiple ADs (multiple pointers) to an object, information about the object that changes at run-time cannot be included in ADs -- it would be too difficult to find and update all the scattered copies of the information. One such changeable attribute of an object is its physical address -- in a dynamic memory system an object can be swapped out to disk and back, or otherwise relocated and have its physical address changed. Because the AD cannot include the physical address of the referenced object, it actually references another descriptor that does, the object descriptor (OD).

There is only one OD for each object, and the OD contains the object's physical address, length, other memory management information, and type information. Object Descriptors are themselves contained in a type of object, an object table object. There can be multiple object tables in a 432 system. One special object table, the object table directory (OTD), contains object descriptors for all other object tables in the system. An access descriptor designates a particular object with a two-part index. The first part selects an object table from the OTD. The second part selects the object's descriptor from the object table.

## SEGMENTS

The most basic objects are segments, contiguous blocks of up to 65536 bytes in memory. An access segment contains only ADs, and is the most basic kind of access object. A data segment contains anything but ADs, and is the most basic kind of data object. Every segment is an object, but not every object is a segment. Objects can be constructed from other objects while segments are the basic material, the simplest kind of object, from which objects are created.

## OBJECT ADDRESSING

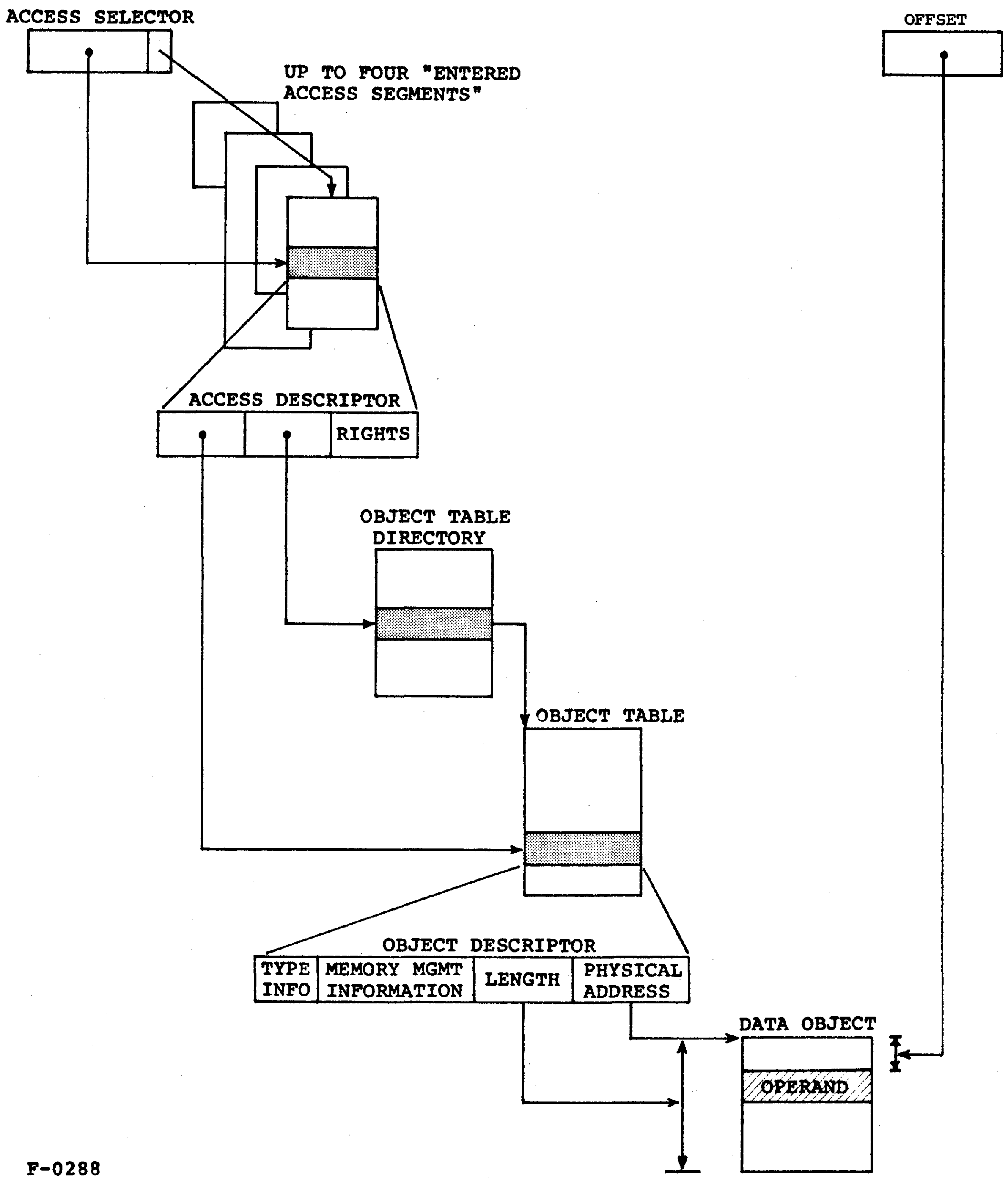
Remember that an executing program is itself represented by an object and can address only objects for which it has ADs. The executing program addresses an object with an index into the lists of access descriptors that it holds.

An executing program can use up to four lists of ADs simultaneously (up to four access segments that in turn specify the objects addressed). The address specified by the executing program is called an access selector. The access selector has two parts. The first part selects one of up to four access segments. The second part selects a particular access descriptor that references a particular object.

For operations on entire objects (such as the 432 operator that sends a message to a "port" object), it is sufficient to specify an object using an access selector. This form of addressing is also all that is needed to specify the operands of operations on access descriptors (such as the 432 operator to copy one AD value over another). The access selector that selects an object also selects a particular AD or AD "slot" in an access object.

For data operations, such as adding or assigning numbers or characters, each data operand is not only in a particular object but at a particular offset into the object. A logical address is thus a two-part address with two independent components, an access selection component that selects a data object and an operand offset component that locates the operand within the data object.

Figure KEY-2 illustrates the chain of descriptors and objects used to address data in a 432 system.



F-0288

Figure KEY-2. Object Addressing

## OBJECT TYPING

Several kinds of type information are associated with 432 objects:

- base type, to determine whether an object contains access descriptors or data
- system type, to determine whether an object is an instance of a system object type defined by the 432 architecture, or is a generic object with no processor-interpreted internal structure
- object descriptor type, to determine whether an object is derived from another object as either a refinement or an extended-type object.

### Base Type

An object's base type can be either access (for an object containing nothing but ADs) or data (for an object containing anything but ADs).

### System Type

An object's system type indicates any processor-interpreted meaning that an object has. An object with system type generic has no processor-interpreted meaning. Other system types designate the system objects used by the 432 for such high-level functions as storage allocation, process scheduling, or interprocess communication.

The interpretation of an object's system type code depends on the object's base type. For example, the same system type code designates either an object table data segment or a domain access segment depending on the base type of the object.

### Object Descriptor Types

There are several types of object descriptors:

- storage descriptors
- refinement descriptors
- type descriptors
- interconnect descriptors

The following sections briefly describe these object descriptors and their use.

## Storage Descriptors

A storage descriptor describes a segment in the 432 storage address space. Such an object can be an access segment or a data segment, can be a generic object or a system object.

## Refinement Descriptors

A refinement descriptor describes an object, called a refinement, that is actually part of another object (see Figure KEY-3). The user of an AD for a refinement can only access the part of the underlying object that is contained in the refinement. For example, if a user is to be allowed access to only part of an employee record, a refinement of the record can be created. Sensitive information such as salary can be excluded from the refinement and cannot be read.

Operands in a refinement are addressed exactly as in a segment, with a displacement from the start of the object (the start of the refinement). A refinement always has the same base type (access or data) as the refined object, but may have a different system type. For example, iMAX uses a generic refinement to give users access to only the fault information area in 432 process objects. A refinement can itself be refined, and there can be multiple different refinements of one underlying object.

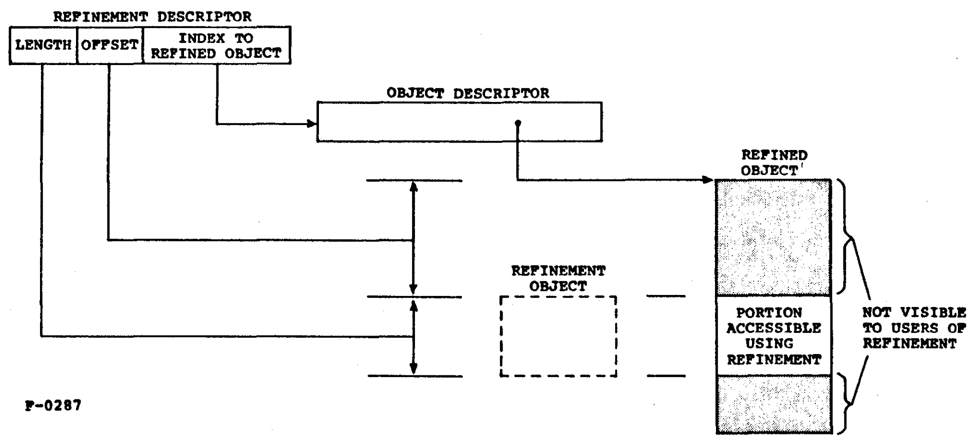


Figure KEY-3. Refinement Object

The 432 architecture uses refinements to implement program modules. Such a module typically has a "public" and a "private" part. The public part is a list of accesses for the operations and data items available to external users of the module. The private part is an access list for internal variables and for other modules used to implement the public operations. The public part of the access list is made a refinement of the entire access list. The caller of an operation can use only the refinement and can reference only the public operations. When the operation is invoked, the processor automatically "traverses" the refinement, to give the context within which the operation executes access to the whole module.

## Type Descriptors

A type descriptor describes an object with a software-defined type, called an extended type object. The type that such an object is a member of is identified by a type definition object (TDO), one of the 432 system objects. A type descriptor references two objects, first the object being typed (called the representation of the extended-type object), and second the TDO that defines the type.

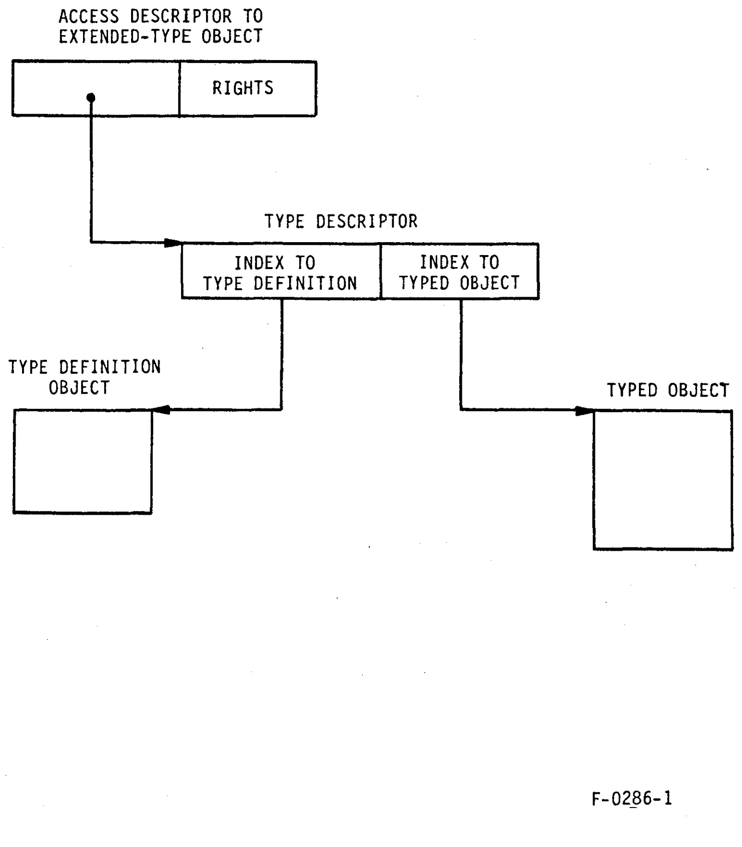


Figure KEY-4. Extended-type Object

### Interconnect Descriptors

An interconnect descriptor describes a kind of data object called an interconnect object. Interconnect objects are contained in a separate interconnect address space. This address space contains hardware registers used for system initialization, interprocessor communication, hardware error reporting, and configuration information.

Interconnect objects do not have a base type or system type. They contain data with no particular interpretation by the processors, and are thus analogous to generic data segments.

Interconnect objects are described here for completeness -- iMAX does not make any interconnect objects visible to users.

## OBJECT CAPABILITIES

Objects support two important 432 capabilities:

- flexible protection of data and program structures
- dynamic storage management

Table KEY-1 gives a tabular summary of the capabilities of 432 objects.

Figure KEY-5 illustrates the three parts of the 432 protection mechanism: "need to know" access control, extensible object typing, and access rights.

Table KEY-1. Object Capabilities

I. "Need to know" Access Control

- An object reference cannot be forged or otherwise corrupted.
- Each activation of a program module has a restricted access environment referencing only those objects that the module activation has a "need-to-know".
- A module can be allowed access to only part of an object by using the object refinement mechanism.

Table KEY-1. (continued)

II. Extensible Object Typing

- Every object has a type and new types can be defined by users.
- Both hardware instructions and software "type manager" modules verify at run-time that object operands are of the proper type.
- Type manager modules can be defined that perform all operations on objects of a particular type. The operations provided by the manager module act as primitives that completely define the behavior of objects of the type. Modules outside the type manager have no access to the internal representation of objects of the type.

III. Access Rights

- The association of access rights with object references allows modules to be granted differing access to the same object (e.g., read-only access for one module and write-only access for another).
- Type-specific access rights allow or disallow operations unique to an object type (e.g., the right to send a message to a port).
- Type manager software can define new access rights, for either system object types or extended types. For example, the iMAX type manager for processes defines a new access right, called "control rights", for processes.

IV. Support for Dynamic Storage Management

- Objects can be relocated in memory (by a "compaction" process) or swapped to disk (in a virtual memory system) because physical addresses for objects are centralized in object tables.
- Because object references are under processor control, dangling references (pointers that outlive the objects they reference) can be prevented and hardware-supported reclamation of no-longer-needed objects is feasible.
- Bounds checking is done on all references to information in objects.

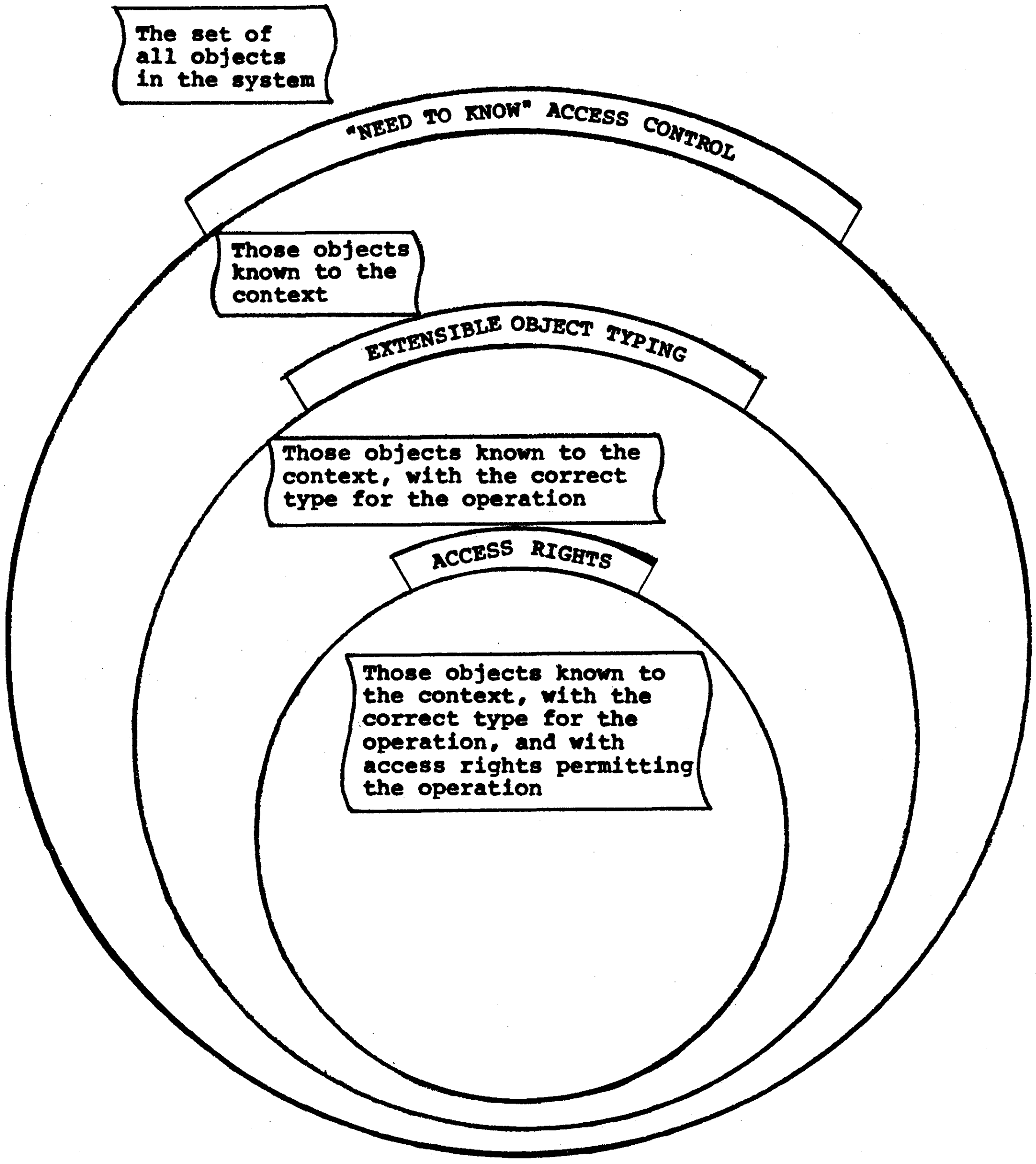
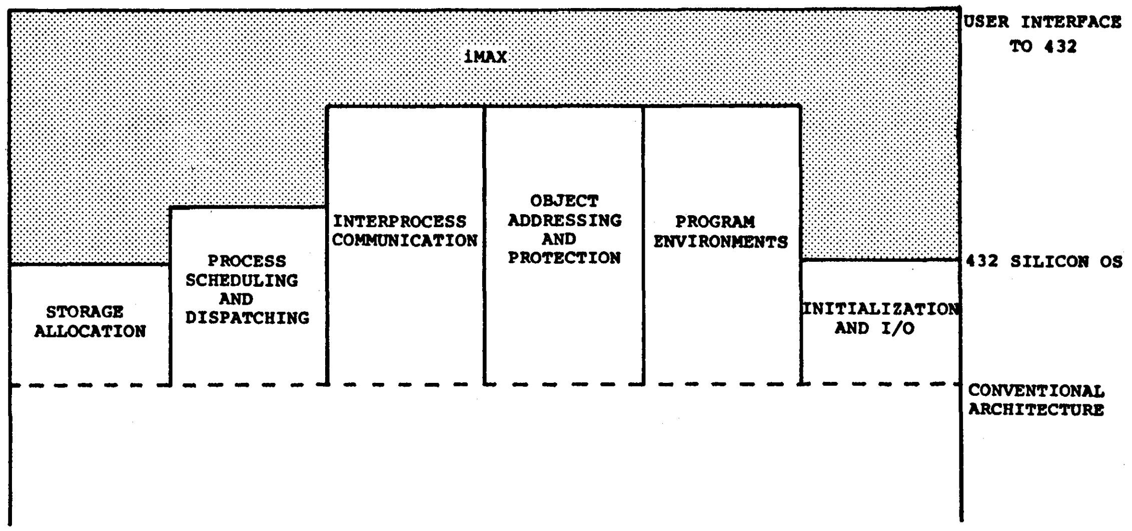

The set of all objects in the system
"NEED TO KNOW" ACCESS CONTROL
Those objects known to the context
EXTENSIBLE OBJECT TYPING
Those objects known to the context, with the correct type for the operation
ACCESS RIGHTS
Those objects known to the context, with the correct type for the operation, and with access rights permitting the operation

Figure KEY-5. Three-fold Object Protection F-0367

## HARDWARE/SOFTWARE PARTNERSHIP

The iAPX 432 architecture provides a higher level of functioning in hardware than conventional computers. Important system structures (e.g., process control blocks and communication buffers) have hardware-recognized representations. High-level operations on these system objects, such as sending a message between processes, are provided as single machine instructions. These features of the 432 are called the "Silicon O.S." These features are not in themselves a complete operating system, but are essential parts of one.

The 432 functions as a hardware/software partnership. Operations are provided in the hardware for any of the following reasons: they are time-critical, thus benefiting from hardware implementation; they are security-sensitive, thus requiring hardware enforcement; or they are complex in a way that benefits from special hardware structures. Other operations are provided by iMAX, cooperating with hardware to provide user-level executive services (see Figure KEY-6).


iMAX
USER INTERFACE TO 432
STORAGE ALLOCATION
PROCESS SCHEDULING AND DISPATCHING
INTERPROCESS COMMUNICATION
OBJECT ADDRESSING AND PROTECTION
PROGRAM ENVIRONMENTS
INITIALIZATION AND I/O
432 SILICON OS
CONVENTIONAL ARCHITECTURE

F-0247

Figure KEY-6. iMAX Complements the 432 Architecture

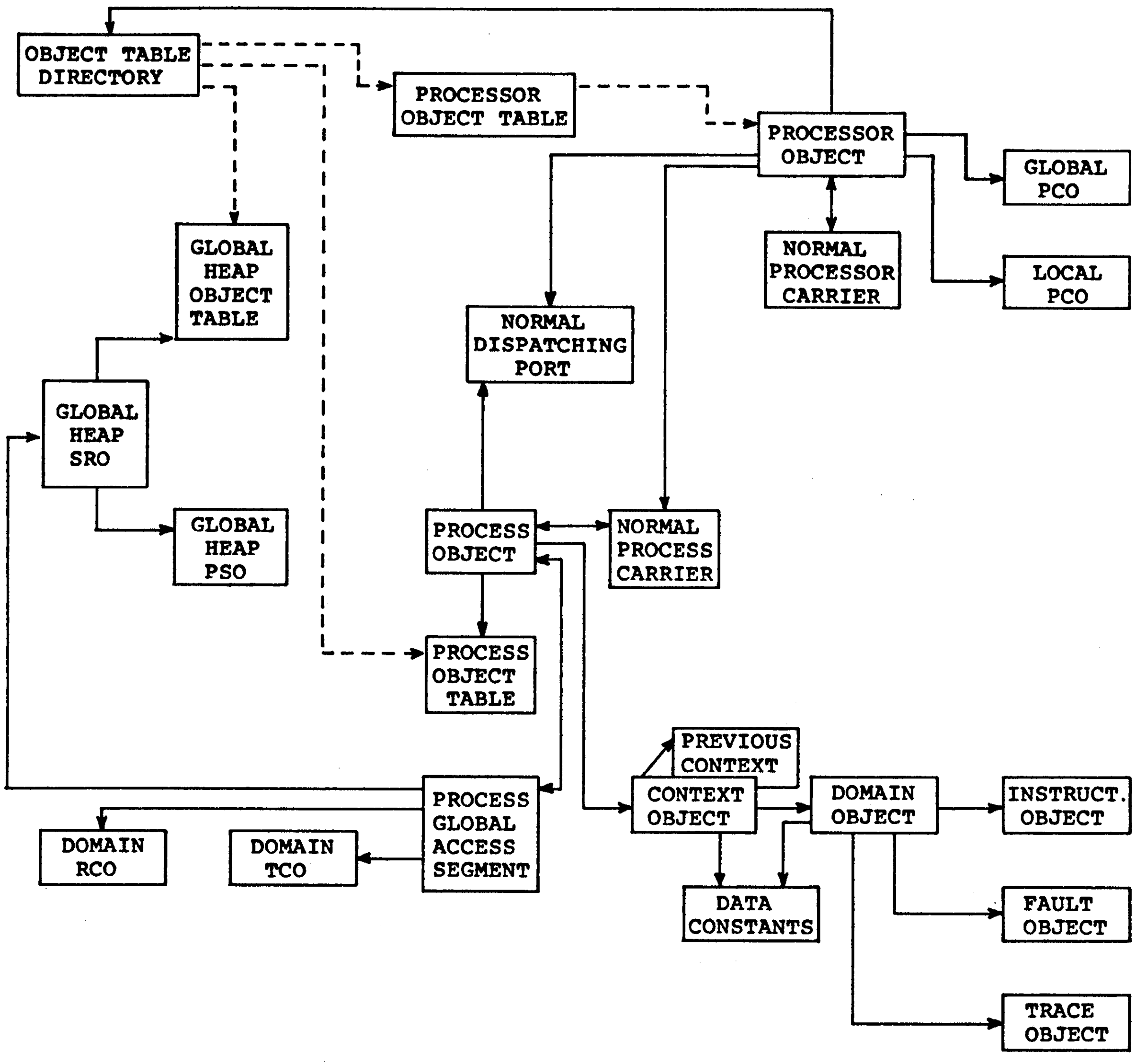
The relationship between iMAX and the hardware is best called "cooperation" because iMAX doesn't simply "run" on hardware that passively executes instructions. The 432 processors act autonomously to provide important services; e.g., a 432 General Data Processor automatically obtains and schedules the next ready process when it needs work. Type-checking and rights-checking are among other services provided by the processors.

Storage management is a good example of the division of labor between iMAX and the 432 processors. The processors recognize system objects used for storage management, provide single instructions that allocate new objects, and set flag bits needed for storage reclamation and virtual storage management. iMAX creates and reclaims local storage pools and provides software processes to compact memory and reclaim unreferenced objects.

Several aspects of the 432 design ensure that the executive structures embedded in silicon are flexible enough for a wide range of applications. First, the hardware and iMAX were designed together, with the high-level services to be provided by iMAX driving the hardware design. Second, care was taken to separate application-specific <u>policy</u> (specified by software-supplied parameters) from general-purpose <u>mechanism</u> (determined by hardware). Third, all 432 system objects can be extended by software, which can define additional object attributes and operations.

## SYSTEM OBJECTS

iMAX is largely a collection of type manager packages for the system objects defined by the 432 architecture. Part of understanding iMAX is understanding the function of these object types and how they work together in a complete system. Note that users do <u>not</u> need to understand the internal details of these objects -- one purpose of iMAX is to make that level of understanding unnecessary. This section describes the system objects that support a 432 system. Figure KEY-7 illustrates some of the system objects required by a system using a minimal iMAX configuration.



| | |
|---|---|
| PCO | = Processor Communication Object |
| PSO | = Physical Storage Object |
| RCO | = Refinement Control Object |
| SRO | = Storage Resource Object |
| TCO | = Type Control Object |

$\longrightarrow$ reference via Access Descriptor
$\dashrightarrow$ reference via Object Descriptor

F-0281

Figure KEY-7. Selected Minimal iMAX Objects

## OBJECT TABLE OBJECTS

Object tables contain object descriptors (ODs), one for each object in a system. User software cannot access object tables directly. User software can obtain an image of the OD for any object that it has access to, but the image cannot be used as an OD. User software can create new objects, which creates new object descriptors. If an object table is full and cannot accommodate new descriptors, iMAX expands it or chains supplementary tables as needed, all transparent to user software.

## PROCESSOR OBJECTS

Processor objects correspond to physical processors in a system, 432 General Data Processors (GDPs) or Interface Processors (IPs). User software cannot access processor objects in any way. User software does specify the number of each type of processor and their hardware identification (i.e., processor ID) as part of configuring an iMAX system.

## PROCESSOR COMMUNICATION OBJECTS

Processor communication objects (PCOs) are used by iMAX to communicate between processors in a 432 system. PCOs are invisible to users.

## PHYSICAL STORAGE OBJECTS

Physical storage objects (PSOs) reference blocks of free memory. User software cannot access PSOs in any way. The user refers to a storage pool via a 432 storage resource object (SRO).

## STORAGE RESOURCE OBJECTS

Storage resource objects (SROs) provide object descriptors and free memory for the creation of new objects. Each SRO references an object table and a PSO. Several SROs can share one PSO. An SRO specifies <u>memory type</u>, <u>lifetime</u>, and <u>reclamation strategy</u> for objects allocated from it. Memory type is frozen or normal. Objects in frozen memory cannot be relocated or swapped and thus are never inaccessible because of memory management operations.

An object's lifetime is either indefinite ("global" objects) or tied to the lifetime of a process or of a subprogram activation within a process ("local" to the process or subprogram activation). When the associated process is destroyed or subprogram activation returns, the object is reclaimed.

An object's reclamation strategy can be based on either garbage collection, or Return from a subprogram activation, or both. Return immediately reclaims all objects whose lifetime is tied to the subprogram activation returned from. Garbage collection is a separate iMAX-provided process that searches out objects that are no longer referenced (and thus no longer used, because the only way to use an object is through a reference to it). When garbage collection finds unreferenced objects, it reclaims both the object descriptor and memory (if any) used by the object. Garbage collection can run at the same time as user processes without interfering with their execution.

There are three types of SROs, corresponding to the three different combinations of reclamation strategies. Stack SROs are used to allocate objects reclaimed only by Return. There is one stack SRO associated with each process object and stack SROs cannot be created separate from process objects. Global heap SROs are used to allocate global objects, which have lifetimes unbounded by any process or subprogram activation. Objects allocated from a global heap can be reclaimed only by garbage collection. Local heap SROs are used to allocate objects that can be reclaimed either by Return or by garbage collection. Such objects have lifetimes tied to some process or subprogram activation and are reclaimed if the associated process is destroyed or associated subprogram activation returns. Also, during their lifetime, garbage collection can reclaim such objects if it finds that they are unreferenced.

Users can hold ADs for heap SROs (global or local) but cannot access the internal representation of SROs. Associated with each process, through its "globals access segment", is an access for a default_global heap_SRO, a heap SRO to be used for creating heap objects if the user does not specify some other heap SRO.

## INSTRUCTION OBJECTS

Instruction objects contain 432 machine instructions. Instruction objects to be called as subprograms also contain a header giving information about the activation record (context) required for the call. Instruction objects are generated by the 432 Ada compiler. Users have read access to instruction objects corresponding to user code.

## DOMAIN OBJECTS

Domain objects correspond to Ada packages and each consists of a list of access descriptors for the elements of the package represented. The public part of a package is represented as a refinement of the corresponding domain. Instruction objects are always called through the domain that references them; the domain is required to establish an access environment for the resulting subprogram activation (context). As part of a subprogram call, the domain refinement is traversed and an access for the entire domain, private and public, is stored in the created context.

## CONTEXT OBJECTS

Context objects correspond to single calls to procedures or functions (to single subprogram activations). A context executes within some domain, called the defining domain, which corresponds to the package in which the subprogram is defined. A context object contains local variables for the subprogram call and is also the root of the access environment for the call. The access environment consists of all objects accessible from the call and includes the defining domain, any constants needed by the call, any parameters passed to the subprogram, and any messages received by the call using interprocess communication. When one subprogram calls another, a chain of context objects results, each referencing its predecessor and with the current context referenced by the associated process.

## PROCESS OBJECTS

Process objects correspond to potentially concurrent activities -- activities that can execute in parallel. The number of processes in a system is the maximum number of activities that can be executing simultaneously. A process is similar to an Ada task, the Ada unit of concurrent activity. Processes can communicate with each other by message-passing. Associated with each process is a stack SRO, consisting of a process object table and an allocation stack for the allocation of contexts and other objects local to the process. Also associated with a process is a processor-recognized process globals access segment (PGAS) that contains information available to all contexts in the process. An example of the information in the PGAS is an access for a default_global_heap_SRO to be used for allocating heap objects.

## PORT OBJECTS

Port objects provide a queuing mechanism for messages and active agents (processors, processes, or surrogates) waiting to send or to receive messages. There are two main types of ports: dispatching ports and communication ports. Dispatching ports are used to queue processor objects waiting for processes to run or to queue process objects waiting for processors to run on. Communication ports are used to transfer messages between processes. Users cannot access dispatching ports, and the unqualified term "port" usually refers to a communication port. A third type of port, a delay port, is used by iMAX to idle processes that suspend their own operation for a specified time period.

## CARRIER OBJECTS

Carrier objects are used as go-betweens for processes and processors using the port mechanism. The purpose of a carrier is to "carry" a message to or from a port. The basic operations on carriers are to send a message in a carrier to a port, or to receive a message in a carrier from a port. Both of these operations can invoke a third operation, the <u>forwarding</u> of the carrier to a second port. One or more carriers are associated with each process and each processor, called <u>process carriers</u> and <u>processor carriers</u> respectively. Both process and processor carriers are invisible to users. Users can create and reference <u>surrogate carriers</u> which can wait to send or receive messages in place of processes, possibly allowing the controlling process to continue doing other work while a surrogate waits in its place.

## TYPE DEFINITION OBJECTS

Type definition objects (TDOs) correspond to software-defined object types, called extended types. An instance of such a type is called an extended-type object. A TDO has an access segment that can reference any attributes of the type. For example, a TDO could reference a domain for a type manager package for the type. Currently, the only attribute that iMAX defines for TDOs is a printable name for the type.

## TYPE CONTROL OBJECTS

Type control objects (TCOs) provide special rights to program modules that reference them (provided the TCO reference itself has certain rights). A TCO access with create rights can be used to create a typed segment, with the type specified by the TCO. A TCO access with amplify rights can be used to amplify specified rights on ADs, optionally restricted to ADs that reference objects with the type specified by the TCO. iMAX provides user software access to only one TCO, for domain objects, so that users can create domains at run-time.

## REFINEMENT CONTROL OBJECTS

Refinement control objects (RCOs) provide special rights to program modules that reference them (provided the RCO reference itself has certain rights). An RCO specifies two object types, one for a refinement and one for the object that is refined. An RCO access with create rights can be used to create a typed refinement. The object being refined must have the type specified by the RCO for the refined object. The resulting refinement has the refinement type specified by the RCO. An RCO access with retrieve rights can be used to retrieve an access for the refined object, given an access for a refinement with the type specified by the RCO. iMAX provides user software access to only one RCO, for domain objects, so that users can create refinements of domains that themselves have type domain.

## MODULAR PROGRAMMING WITH ADA

The iMAX interface seen by users is specified in Ada, and iMAX takes advantage of Ada's support for modular programming and compile-time checking, both in its user interface and in its internals.

The iMAX interface is a collection of Ada packages that each provide one well-defined area of function, typically as a type manager for one system object type. iMAX appears to users more as a subprogram library than as a monolithic entity -- users need only concern themselves with the packages they are using. The Ada user also benefits from compile-time checking of all calls to iMAX services.

There is no distinction between iMAX packages and user-written packages. iMAX operations and user operations are invoked in the same way. There is no special calling convention, no "Supervisor Call" instruction, and no need to drop into assembly language to invoke the executive.

iMAX's organization as a library of packages allows users to:

- create subsets of iMAX -- by omitting unwanted packages
- create supersets of iMAX -- by adding their own packages
- create variations of iMAX -- by providing their own implementation for iMAX interfaces

## PACKAGES

An Ada package is a named collection of types, data, and/or subprograms in two parts:

- specification -- defines the interface seen by a user of the package
- body -- defines data and/or subprograms used only within the package body and gives the code that implements any subprograms defined in the specification. The body can also contain initialization code for the package.

The specification is itself in two parts:

- public part -- defines the logical interface seen by a user of the package
- private part -- provides declarations and representation details that the Ada compiler requires to use the specification. However, these declarations and details cannot be referenced by users of the package and can be treated as part of the package implementation.

The body and private part of a package can be omitted if they would otherwise be empty.

The modular programming example in the next section illustrates the syntax for all parts of a package.

## A MODULAR PROGRAMMING EXAMPLE -- IMPLEMENTING SEMAPHORES

This section gives an example of modular concurrent programming using iMAX and Ada. The example is realistic and intended to solve a real problem. In reading and understanding the example, you will encounter many of the Ada idioms used in this manual. A large part of reading this manual is reading and understanding Ada packages; the example is a taste of things to come.

The example is an Ada package that acts as a type manager for binary semaphores. Binary semaphores are frequently used in concurrent programming to ensure that only one process at a time is using some resource (e.g., only one process at a time is writing to a particular record in a data base). Semaphores are sometimes called locks.

The example shows how iMAX can be extended to provide services not yet supported directly. The example also illustrates how Ada separates the specification and implementation of a package.

## SEMAPHORES PACKAGE

```
with iMAX_Definitions;
package Semaphores is

  -- Function:
  --   This package is a type manager that defines binary semaphores and
  --   the operations on them.

  --   A "binary semaphore" is a flag used by multiple processes that
  --   must share some resource, where only one process at a time can
  --   use the resource.  An example is two processes that share a
  --   printer for producing reports.  While process A is using the
  --   printer, process B should not write to the printer -- or output
  --   from process B could appear jumbled in a report from process A.
  --   A binary semaphore can be used, by convention, to provide
  --   "mutual exclusion" between processes using the printer.  A
  --   semaphore has two states, AVAILABLE and IN_USE.  When a process
  --   wants to use the printer, it calls the Get operation on the
  --   associated semaphore.  If the semaphore is AVAILABLE, Get
  --   indivisibly sets it to IN_USE.  If the semaphore is IN_USE by
  --   another process, the process calling Get is blocked until the
  --   semaphore is AVAILABLE.  In either case, Get does not return
  --   until the semaphore is IN_USE by the calling process.  When the
  --   process has finished using a resource (the printer), it calls the
  --   Release operation on the associated semaphore.  Release makes the
  --   semaphore AVAILABLE for the next use of the resource, either by
  --   the same process or by another process.  If processes are waiting
  --   for the semaphore when Release is called, Release can also simply
  --   leave the semaphore IN_USE and unblock the first waiting process.

  --   This manager for semaphores requires that the process releasing a
  --   semaphore either be the same process that got use of it, or
  --   supply an access with control rights for the process that is
  --   using the semaphore.

  --   The operation Try is provided which gets a semaphore only if it
  --   is available immediately, and returns an indication of success
  --   or failure.

  --   The function Value returns whether the semaphore is AVAILABLE or
  --   IN_USE.  This function can be useful for sampling semaphore
  --   usage.

  --   The function User returns an access (without access rights) for
  --   the process using a semaphore, and returns null if the semaphore
  --   is available.
--   Two exceptions are defined, get_error and release_error.
--   Get_error is raised if a process tries to get a semaphore that
--   it already is using.  Release_error is raised if a process tries
--   to release a semaphore that is AVAILABLE, or tries to release a
--   semaphore that is being used by another process and the
--   releasing process does not present an access with control rights
--   for the using process.

-- Notes:
--   The names used for operations in this package are generic (e.g.,
--   "Create" rather than "Create_semaphore") and should be qualified
--   with the package name when invoked, e.g.:

--     my_sem := Semaphores.Create();  -- correct usage
--
--     my_sem := Create();             -- questionable usage

--   This specification leaves undefined the order in which waiting
--   processes get a semaphore.  Programs that rely on a particular
--   ordering are erroneous.

--   When a semaphore is used to synchronize use of a shared resource,
--   it is best to embed the semaphore and its use in the private part
--   or body of a package which manages the resource.  So long as the
--   manager package is the only way to access the resource,
--   synchronization of use is guaranteed.  This approach centralizes
--   use of the semaphore, reduces the complexity of the application
--   processes, and eliminates the possibility of a bug caused by an
--   applications process not following the convention of using the
--   semaphore to control access to the resource.

--   There are options in managing semaphores that are not supported
--   by this package, but not excluded by it either.  Users can
--   implement some of these options in higher-level managers that
--   call on this one.
```

```
use iMAX_Definitions;


type semaphore_rep is limited private;   -- conceal representation

type semaphore is access semaphore_rep;
                             -- Each semaphore is a distinct 432 object.

type state is (available, in_use);


get_error:      exception;  -- raised if process tries to get a
                            -- semaphore that it is already using
release_error:  exception;  -- raised if a process tries to release a
                            -- semaphore that is available or tries to
                            -- release a semaphore in_use by another
                            -- process without presenting an access
                            -- with control rights for the using
                            -- process


function Create(s: state := available)  -- initial state
  return semaphore;

  -- Function:
  --   Create a semaphore and return an access for it.  The default
  --   initial state is available with no processes waiting for the
  --   semaphore.  If the caller specifies that the semaphore is
  --   initially in_use, then the calling process is also the initial
  --   user process.

  -- Note:
  --   The setting of the access rights bits on the returned AD are
  --   undefined by this specification.


procedure Get(sem:  semaphore);

  -- Function:
  --   Return only when the calling process has exclusive use of the
  --   semaphore.  Block the calling process until then if necessary.
  --   On return, the state of the semaphore will be in_use and the
  --   calling process will be the semaphore's using process.

  -- Exceptions:
  --   get_error        -- raised if a process calls Get on a semaphore
  --                    -- that it is already using
```

```
procedure Release(
    sem:  semaphore;
    user: process := null); -- If defaulted to null, calling
                               -- process must be user.

  -- Function:
  --   If the semaphore is already available, raise release_error.

  --   Otherwise, if the specified user is not the actual user,
  --   raise release_error.

  --   Otherwise, if the user is not the calling process and the user
  --   access parameter does not have control rights, raise
  --   release_error.

  --   Otherwise, make the semaphore available to the next process
  --   requesting it.  If blocked processes are waiting for the
  --   semaphore, Release can simply unblock one of the waiting
  --   processes and leave the semaphore in_use by the new user
  --   process.

  -- Exceptions:
  --   release_error


procedure Try(
    sem:        semaphore;
    succ:   out boolean);

  -- Function:
  --   Get a semaphore only if it is immediately available (i.e.,
  --   only if the calling process need not block and wait for it).
  --   The parameter succ is assigned true if the semaphore is
  --   gotten, else false.

  -- Exceptions:
  --   get_error        -- raised if the semaphore is already in_use by
  --                    -- the calling process.


function Value(sem:  semaphore)
  return state;

-- Function:
--   Return whether the semaphore is available or in_use.
```

```
  function User(sem:  semaphore)
    return process;

    -- Function:
    --   If the semaphore is available, return null.  Otherwise the
    --   semaphore is in_use and return an access for the using
    --   process.


private

  type semaphore_rep is record
    port:  iMAX_Definitions.port;
    user:  process;
  end record;


  pragma inline(Create, Get, Release, Try, Value, User);

end Semaphores;
```

SEMAPHORES PACKAGE BODY

```
with Descriptor_Definitions, iMAX_Definitions,
     Basic_Process_Management, Untyped_Ports,
     Process_Globals_Definitions;
package body Semaphores is

  -- Logic:
  --   The semaphore uses a port with one queue entry.  If the
  --   port is empty, the semaphore is available.  A process gets
  --   the semaphore by sending a message to the port.  If the
  --   port is full, the requesting process blocks until the process
  --   using the semaphore releases it by receiving from the port.

  use Descriptor_Definitions, iMAX_Definitions, Untyped_Ports,
      Process_Globals_Definitions;

  package BPM renames Basic_Process_Management;

  function Create(s: state := available)  -- initial state
    return semaphore is

    sem:  semaphore;

  begin
    sem := new semaphore_rep(
            port => Create_port(1),
            user => null);

    if s = in_use then
      sem.user := Process_globals().owner; -- access for calling process
      Send(sem.port, any_access(sem.user));
    end if;

    return sem;
  end Create;


  procedure Get(sem:  semaphore) is

    p:  process;

  begin
    p := Process_globals().owner;
    if sem.user = p then
      RAISE get_error;

    else
      Send(sem.port, any_access(p));
      sem.user := p;
    end if;

    return;
  end Get;
```

```
procedure Release(
    sem:        semaphore;
    user:       process := null) -- If defaulted to null, calling
  is                             -- process must be user.

  p:  process := user;
  succ: boolean;

begin
  if p = null then
    p := Process_globals().owner;
  end if;

  if sem.user /= p then
    RAISE release_error;

  elsif user /= null and then
        not Permits(user, BPM.control_rights) then
    RAISE release_error;

  else
    sem.user := null;
    Cond_receive(sem.port, any_access(sem.user), succ);
    if not succ then
      RAISE release_error;

    else
      RETURN;

    end if;
  end if;
end Release;


procedure Try(
    sem:        semaphore;
    succ:   out boolean) is

  p:  process;

begin
  p := Process_globals().owner;
  if sem.user = p then
    RAISE get_error;

  else
    Cond_send(sem.port, any_access(p), succ);
    if succ then
      sem.user := p;
    end if;
    RETURN;

  end if;
end Try;
```

```
  function Value(sem:  semaphore)
    return state is

  begin
    if sem.user = null then
      RETURN available;

    else
      RETURN in_use;

    end if;
  end Value;


  function User(sem:  semaphore)
    return process is

  begin
    return sem.user;
  end User;

end Semaphores;
```

# CHAPTER HOW
# HOW TO USE iMAX

This chapter introduces new users of iMAX to techniques for writing Ada programs that call on iMAX services:

- How to reference iMAX services

- How to convert values to and from the types used by iMAX

This chapter does not describe how to use the iMAX AP software. The AP software is written in PL/M-86 and is described in Chapter IOI, Input/Output Implementation.

None of the information in this chapter is specific to iMAX. It is a review of material defined by the Reference Manual for the Ada Programming Language and by the Reference Manual for the Intel 432 Extensions to Ada.

## HOW TO REFERENCE iMAX SERVICES

iMAX is organized as a collection of Ada packages. To reference a particular subprogram in iMAX, you must include in your own module certain references to the package containing the subprogram:

- an environment pragma that gives the file name of the environment file for the iMAX package.

- a with clause that declares the Ada name of the iMAX package.

- optionally, a use statement in your module, that makes the elements of the iMAX package directly visible in your module without being prefixed with the package name.

All three constructs can reference multiple packages.

## THE ENVIRONMENT PRAGMA

An environment pragma informs the compiler of the names of the environment files that constitute an environment for compiling a package. The pragma must appear as the first Ada construct (excluding comments) in your source file, and can appear only once in a file. The environment files named in the pragma must include environments for all package specifications named in with clauses in the source file. The environment pragma must also include environments for the bodies of any generic packages or inline subprograms used. Also, the environment pragma for a package body must reference the environment for the associated package specification. The arguments to the environment pragma are character strings, enclosed in double quotes. Note that the Ada package Standard is implicitly a part of every compilation environment and need not be explicitly referenced. Appendix SUM, Software Components Summary, lists the environment files that correspond to all user-visible iMAX packages.

## THE WITH CLAUSE

A with clause is an Ada context specification that appears as a prefix to a package specification or body. The with clause lists the Ada names of any separately compiled packages referenced by the following package. The packages named by the with clause must also have their environments referenced by the preceding environment pragma.

## THE USE STATEMENT

A use statement names packages that the programmer wants to be directly visible. For example, suppose that a programmer has referenced a package Other_Package in an environment pragma and with clause. At this point, a routine Some_routine in Other_Package can be referenced by qualifying each reference with the package name, e.g., "Other_Package.Some_routine". However, by naming Other_Package in a use statement, "use Other_Package;", the contents of Other_Package are made directly visible, and the programmer can invoke Some_routine without qualification. A use statement can appear in the declarative portion of a package or subprogram.

The benefit of the use statement is the brevity of references that it allows. The drawback is that the resulting code may be less maintainable, because it is more difficult to identify references to external packages. In the examples in this manual, a balanced approach is taken. A use statement names the most commonly used external packages but references to less frequently used packages are explicitly qualified. Of course, even when a use statement is in your program, you must still qualify ambiguous references. For example, if two packages A and B both contain a subprogram Read, then you must identify a call to Read as A.Read or B.Read. Note that the Ada package Standard is implicitly used in every compilation and need not be explicitly referenced by a use statement.

## AN EXAMPLE

For example, suppose that you want to write a package with a single procedure Main that writes the message "HELLO WORLD" to the Intel 432 Cross Development System console. Reading Chapter IO, Input/Output, you find that the Debug_Sink package provides the procedure Write that writes a message to the console. Looking in Appendix SUM, Software Components Summary, you find that the file name for the environment for the Debug__Sink package is "DBISNK.MLE". First you write your specification:

```
package Hello_World is

  -- Function:
  --   Say "HELLO WORLD" to the debug console via procedure Main.

  procedure Main;

    -- Function:
    --   Say "HELLO WORLD" via the debug console interface.

end Hello_World;
```

You might call your source file HELLO.MSS. This is your package specification, and does not need to reference the Debug_Sink package. The Debug__Sink package is visible only from the body (from the implementation) of your Hello_World package. Therefore, your specification has no environment pragma, no with clause, and no use statement. Now you write your body:

```
pragma environment(
  "acs:maxdef.mle",
  "acs:dbisnk.mle",
  "acs:cnvrt.mle",
      "hello.mse");

with iMAX_Definitions, Debug_Sink, Unchecked_Conversion;
package body Hello_World is

  procedure Main is

    line_length:  constant integer := 80;
    type buffer_rep is new string(1..line_length);
    type buffer is access buffer_rep;
    function Retype_buffer_to_any_ds
      -- (b: buffer)
      -- return any_ds;
      is new Unchecked_conversion(buffer, iMAX_Definitions.any_ds);

      -- Function:
      --   The given access for a buffer is retyped to any_ds.
      --   This operation does not affect the rights on the access.


    line:  buffer := new buffer_rep;

  begin
    line(1..11) := "HELLO WORLD";
    line(12) := ASCII.lf;  -- defined by Ada in Standard package;
                           -- linefeed is iMAX end-of-line character.

    Debug_Sink.Write(Retype_buffer_to_any_ds(line), 0, 12);
  end Main;
end Hello_World;
```

As shown, the environment pragma for the body references the environments for the iMAX_Definitions, Debug_Sink, and Unchecked_Conversion units and for the Hello_World package specification. The with clause for the body references the Debug_Sink, iMAX_Definitions and Unchecked_Conversion units referenced in the body.

There is no use statement in the body, so the Write procedure in Debug Sink must be referenced as Debug_Sink.Write. Note that there is a reference to the component ASCII.lf in the Ada Standard package without prefixing the reference with the package name. This reference is valid because the Ada Standard package is implicitly used in every compilation.

## HOW TO CONVERT VALUES TO AND FROM THE TYPES USED BY iMAX

Ada is a strongly-typed language. Doing type conversion is a necessary part of using iMAX. Figure HOW-1 gives examples of how to convert to and from the type any_access defined by the Intel 432 Extensions to Ada.

```
-- Definitions used in the example
   type my_record is limited private;
   type my_access is access my_record;
   my_access_variable:   my_access;
   any_access_variable:  any_access;

-- You can convert a value of your access type to the any_access type
-- like this:
   any_access_variable:= any_access(my_access_variable);

-- To convert from any_access back to your access type, you must first
-- define a function to do the conversion that is an instantiation of
-- the generic Ada function Unchecked_Conversion:

   Function Retype_any_access_to_my_access
   -- (a: any_access)          --these comment lines show form
   -- return my_access;        --of new function.
   is new Unchecked_conversion(any_access, my_access);

-- OK, now use the new function.
   My_access variable :=
     Retype_any_access_to_my_access(any_access_variable);
```

Figure HOW-1. Conversion Example

The techniques of Figure HOW-1 for conversion between access types can be used for other types as well. Note however, that the 432 architecture does not allow unchecked conversions between an access type and a non-access type. Refer to the Reference Manual for the Ada Programming Language for more information on type conversions in Ada.

# CHAPTER DEF
# BASIC DEFINITIONS

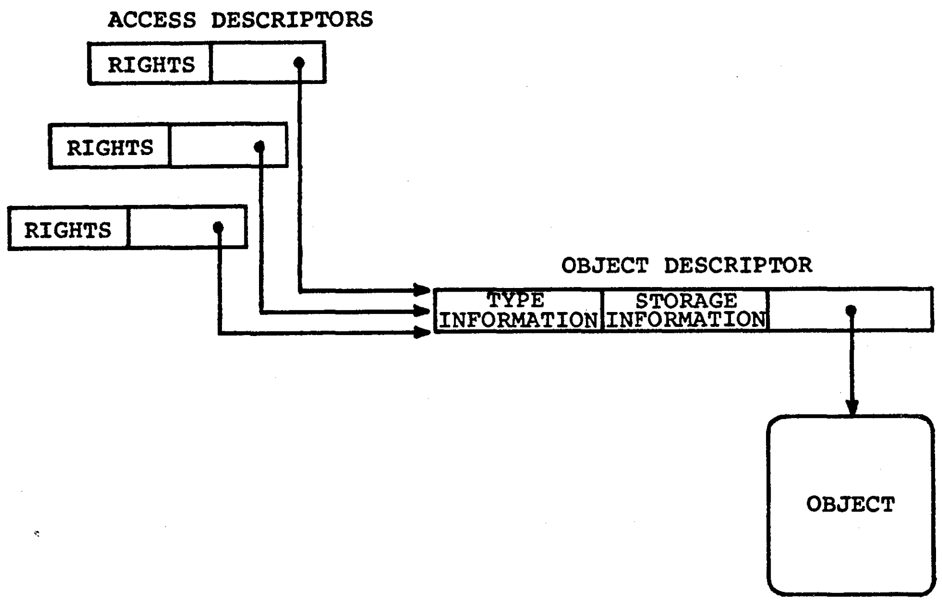
This chapter describes a common set of constants, data types, and operations used throughout iMAX and available to iMAX users.

## DESCRIPTOR DEFINITIONS

The Descriptor__Definitions package provides an Ada interface to the 432's object addressing mechanism (Figure DEF-1):

- access rights and AD rights associated with 432 access descriptors (ADs, Ada access values)
- type and storage information associated with 432 object descriptors (ODs)

Operations are provided to check and remove rights on access descriptors, to check type information in object descriptors, and to inspect access descriptors and object descriptors.



F-0033

Figure DEF-1. 432 Object Addressing

## RIGHTS

Descriptor_Definitions defines rights on access descriptors and operations to test and to remove rights on ADs. There are no visible iMAX operations to add rights to ADs; iMAX can only be used to restrict, not increase, rights. Rights on ADs consist of access rights and AD rights. AD rights are to the AD itself and consist of delete rights, which allow the AD to be overwritten, and unchecked copy rights, which allow the AD to be copied without a level check.

Level checking keeps an object reference from being copied into an object with a longer lifetime than the referenced object. Such copying of a reference could create a "dangling reference" when the referenced object is deleted before the reference to it. A reference to a deallocated object would be a protection violation, an AD value that could be used later to reference an entirely different object when either the object descriptor or the physical memory associated with the deallocated object was reused. Level checking is automatically suppressed for references to objects allocated from global heaps; such objects have indefinite lifetimes and are only deallocated when no more references exist for them. Otherwise, the right to bypass level checking is used only within iMAX in certain special circumstances, and is not available to users.

Access rights restrict operations on an object via a particular AD. Access rights consist of representation rights and type rights. Representation rights consist of the primitive read and write rights applicable to any segment or refinement. Type rights restrict the right to execute certain operations using an AD; those operations depend on the type of the referenced object. Type rights are typically renamed for each object type that uses them. For example, for port objects, send rights and receive rights rename the first two type rights and restrict the rights to send or to receive messages using a port access. There are three type rights. For any given object type, hardware may interpret from zero to three type rights. Uninterpreted type rights can be used by software. Appendix RGT, User-Visible Type Rights, tabulates iMAX-defined type rights.

Descriptor_Definitions defines all these rights as constants of the type rights and also defines the placeholder value no such right. The function Permits can be used to check for any combination of rights in an access descriptor. Permits always returns false when given a null AD. The procedure Remove can be used to remove any combination of rights from an AD.

## OBJECT TYPES

432 objects are system objects, generic objects, or extended-type objects. Extended-type objects have a type supplied by software. System objects and generic objects have a three-part hardware-recognized object type, consisting of base type, system type, and processor type. The constants of the form xxx type rep (see page DEF-11) define the values of these type fields, and can be used with the hardware type pragma defined by the 432 extensions to Ada (see Warning).

**WARNING**

The hardware type pragma should not be used while also using iMAX. iMAX defines type managers for many types of system objects. A user creating system objects independently may conflict with programming conventions obeyed by the iMAX type managers, resulting in an erroneous program.

Four boolean functions provide object type-checking. Check base type returns true if an object has a specified base type. Check system type returns true if an object has the base type and system type of a specified object type (the processor type field is ignored). Check object type returns true if an object has a specified object type (base, system, and processor types -- see Figure DEF-2). All three functions return true if passed a null access, and false if passed an access for an extended-type object. The fourth function, Check extended type, returns true if passed a null access or an access for an extended-type object. All of these functions return true when passed a null access so that they can be used to check for membership in access subtypes that include null.

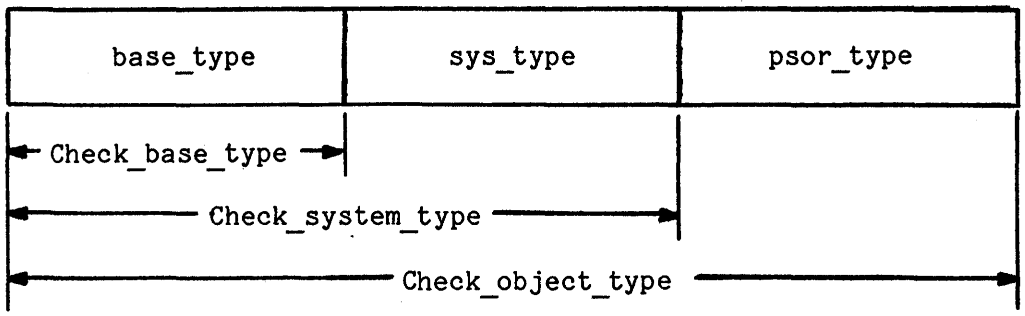


Figure DEF-2. object_type Fields and Checking Functions

## MISCELLANEOUS

Descriptor_Definitions also includes constants and types for the maximum size of physical memory, for the maximum segment size, and for object table indexing.

## INSPECTING DESCRIPTORS

The Inspect access function returns an access descriptor represented as an ac descr record. The Inspect object function returns a record containing both a representation of the access descriptor, and, if the access descriptor is not null, a representation of the referenced object descriptor.

These representations are useful only for inspecting the information contained in the descriptors. The record representation for object descriptors is complex, partially because variants are included which are used only by iMAX and are of no utility to users. Only storage, refinement, and type descriptors are normally visible to users. Table DEF-1 provides a convenient cross reference of which fields are defined for which object table entry types. More information about the type and meaning of these object descriptor fields can be found in the iAPX 432 General Data Processor Architecture Reference Manual (GDP ARM). The only fields that are not described by the GDP ARM are the three "white" bits, rwhite, swhite, and twhite, which are defined by iMAX software for use by the iMAX garbage collection process.

Table DEF-1. Object Table Entries Cross Reference

| | ENTRY TYPE: | | | | | |
|---|---|---|---|---|---|---|
| FIELD NAME: | Header | Free | Interconnect | Storage | Refinement | Extended |
| descr_type | X | X | X | X | X | X |
| descr_subtype | X | X | X | | | |
| free_index | X | | | | | |
| end_index | X | | | | | |
| fault_level | X | | | | | |
| cur_level | X | | | | | |
| stor_claim | X | | | | | |
| next_free | | X | | | | |
| valid | | | X | X | X | X |
| reclamation | | | X | X | X | X |
| level | | | | X | X | X |
| length | | | X | X | X | |
| local_adr | | | X | | | |
| stor_assoc | | | | X | | |
| io_lock | | | | X | | |
| altered | | | | X | | |
| accessed | | | | X | | |
| base_adr | | | | X | | |
| dirty | | | | X | | |
| swhite | | | | X | | |
| prv_level | | | | X | | |
| base_type | | | | X | X | |
| sys_type | | | | X | X | |
| psor_type | | | | X | X | |
| byp_ot_index | | | | X | | |
| byp_dir_index | | | | X | | |
| rwhite | | | | X | | |
| base_disp | | | | X | | |
| rfn_obj_ref | | | | X | | |
| privat | | | | | X | |
| twhite | | | | | X | |
| typed_def_ref | | | | | X | |
| typed_obj_ref | | | | | X | |

NOTE: X = field is present in entry

The object descr record includes such useful information as the object's length (where applicable) and lifetime (given by the level number). The user should refer to the iAPX 432 General Data Processor Architecture Reference Manual for more information about the use and meaning of the object descriptor fields.

## iMAX DEFINITIONS

The package iMAX_Definitions contains miscellaneous definitions needed by users:

- access subtypes corresponding to hardware object types
- a package type used to pass procedures as values
- default process scheduling parameters
- generic views of access segments and data segments
- the access_selector data type
- the Idle procedure, to block a calling process for a specified time
- the print_name data type, used to associate symbolic names with objects.

### ACCESS SUBTYPES

iMAX_Definitions defines access subtypes corresponding to the 432 base types, to the system types of all 432 system objects, and to 432 extended-type objects. For example, the access subtype port is defined as:

```
    subtype port is any_access;
```

The elements of the port subtype are accesses for objects with system type port as (port access segment) and base type as (access segment).

The correspondence between the access subtypes and the types of the objects referenced is implicit and is enforced in two ways. First, the iMAX package that acts as a type manager for ports requires that all parameters that reference port objects be of type port. Second, and more important, all the operations on ports available to the iMAX user are type-checked by hardware. Any attempt to use an access for an object other than a port where a port access is required will raise the Ada constraint_error exception.

Some of the access subtypes defined are used only within iMAX and should not be used by users. Users do not have access to values of these types:

| ACCESS SUBTYPE: | REFERENCES: |
|---|---|
| object_table | object table data segment |
| process_control | process data segment |
| port_control | port data segment |
| carrier_control | carrier data segment |
| processor | processor access segment |
| processor_control | processor data segment |
| processor_communication | processor communication data segment |
| physical_storage | physical storage data segment |

The access subtypes any as and any ds correspond only to the base types as (access segment) and ds (data segment) with no constraint on system type or processor type. By contrast, the access subtypes access sgt and data sgt correspond to the object types for generic access or data segments accessible by all processors.

The access subtype extended type includes all accesses for extended-type objects. These objects do not themselves have any hardware object type but consist of two references, one for a type definition object, and the other for the representation of the extended-type object (which can have a hardware object type). Operations on extended-type objects are described in Chapter EXT, Extended Types.

All of the other access subtypes given correspond to a particular base type and system type for the referenced objects (as in the port example above).

Note that all access subtypes include the access value null.

## PROCEDURE VALUE

iMAX_Definitions defines the package type procedure_val which matches any package or package refinement with a public part consisting of one procedure with no parameters.

iMAX uses procedure_val as the type of the initial procedure parameter for creating static processes. iMAX static processes and how they are created are described in Chapter CON, Configuration.

The package type iMAX_Definitions.procedure_val should not be confused with the similar package type Basic_Process_Management.procedure_val, which is used in the creation of dynamic processes and is described in Chapter BPM, Basic Process Management.

Package types are described in the Reference Manual for the Intel 432 Extensions to Ada.

## DEFAULT PROCESS SCHEDULING PARAMETERS

iMAX_Definitions defines the default scheduling parameters for iMAX static processes (described in Chapter CON, Configuration). These scheduling parameters can also be used for iMAX dynamic processes (described in Chapter BPM, Basic Process Management). If used for dynamic processes, these scheduling parameters must be explicitly specified when the dynamic process is created. Because user processes compete with iMAX system processes for scheduling, users of scheduling values other than the default ones should consult Appendix PRS, Process Scheduling Information.

The process scheduling parameters rely on the concept of a system time unit determined by hardware. For 432/600 systems, a system time unit is 256 microseconds.

service period is the maximum number of system time units that a process can run before it is rescheduled, giving other processes a chance to run. default service period is 400 system time units (about 0.1 seconds on 432/600 systems).

deadline is the relative urgency of a process at the time it is scheduled; a smaller deadline indicates greater urgency. When a process is scheduled, its absolute deadline value is computed by adding the deadline scheduling parameter to the current system time. The effect of this adjustment is that the "urgency" of a process increases as it waits to be dispatched. Processes are scheduled in order of increasing absolute deadline within priority. Because of the deadline parameter, a process that has been waiting for a long time for a processor can be scheduled ahead of a usually more urgent process that has just been scheduled. For example, consider two processes, A and B, with the same priority. If process A has deadline parameter 800 and has been waiting for 600 system time units, it is scheduled ahead of process B which has deadline parameter 300 (normally more urgent) but has just now been scheduled. One way to understand the deadline parameter is to consider it to be the time that a process can wait for dispatching -- more urgent processes are less able to wait. All other things being equal, a process with a smaller deadline value will spend a shorter time waiting in the dispatching queue to run than a process wih a larger deadline value. default deadline is 1600 system time units (about 0.4 seconds on 432/600 systems).

priority is an absolute measure of the relative importance of competing processes -- if processes with different priorities are all ready to be dispatched, then the higher-priority process is dispatched first. Note that a higher priority process that has just become ready will run before a lower priority process that has been waiting for a long time, i.e., a higher-priority process can "starve" lower-priority ones. Priority is a short ordinal with 0 being the lowest priority and 65535 being the highest priority. default priority is 10.

## GENERIC VIEWS OF SEGMENTS

Some software applications require a view of segments not as objects with particular types and attributes, but as a collection of access descriptors or of bytes:

- A "garbage collection" program traverses a network of segments to mark all segments reachable from some set of roots. The program views access segments purely as collections of access descriptors and ignores whatever meaning may be associated with them.

- Debugger software can provide a display command that views the bytes in a data segment in hex or ASCII, without regard for higher-level meanings.

- iMAX input/output views I/O in terms of streams of bytes in chunks called buffers. The only structure recognized in the data is arrays of bytes (characters), though higher-level software may view those same bytes as numbers, dates, or data-base transactions.

iMAX_Definitions defines the type any access array as an access to an array of any_access values with index range 0..16383.

iMAX_Definitions defines the type byte array as an access to an array of bytes with index range 0..65535. The type byte is defined in the 432 extensions to Ada as a new entry in the Ada package STANDARD:

```
    type byte is new ordinal range 0..255;
```

Both array types are fixed arrays corresponding to maximum segment size. However, the 432 hardware will detect any reference beyond the bounds of any segment.

iMAX_Definitions also defines the functions Retype to any access array and Retype to byte array which perform unchecked conversion from types any_as (any access segment) to any_access_array and any_ds (any data segment) to byte_array.

## ACCESS SELECTORS

432 access selectors are part of the hardware addressing mechanism and are not normally visible to users programming with Ada. Access selectors are visible to users who use iMAX to examine context objects, instruction objects, or fault information, as described in Chapter PEN, Program Environment Access.

An access selector selects one access descriptor from the immediate access environment of a context. This access environment consists of up to four entered access segments (EASs), designated EAS 0 up to EAS 3. EAS 0 is always the context access segment of the current context.

An access selector value has two fields; the EAS field selects one of the entered access segments, and the index field selects an AD in the selected segment.

Some access_selector values are used with a second interpretation, as a domain access index. In this interpretation, the EAS field is ignored and the index field selects an AD in the defining domain of the current context (the domain specified in the Call operation that created the context).

## IDLE PROCEDURE

iMAX_Definitions provides the procedure Idle to allow a process to block its own execution for at least a specified number of system time units (a system time unit is 256 microseconds for 432/600 systems). Note that Idle(0) simply reschedules the calling process in the dispatching queue.

## PRINT NAMES

For use in error messages and debugging, iMAX supports the association of printable names with some system objects. To make these names both efficient and uniformly usable, iMAX_Definitions defines the type print name. This character array is used by iMAX type managers to provide symbolic names for these types: processes, type_definitions, query records (one per synchronous I/O interface), and connections (one per asynchronous I/O interface). Using print_names is not a secure way to identify objects because multiple objects can have the same name. Their major use is in debugging OEM system software.

Note that values of type print_name must be exactly 15 characters:

```
    "MY_PROCESS"       -- wrong

    "MY_PROCESS     " -- right
```

## DESCRIPTOR DEFINITIONS PACKAGE

```
package Descriptor_Definitions is

  -- Function:
  --   This package contains types and operations related to access
  --   descriptors and object descriptors.


  -- RIGHTS  -  RIGHTS CHECKING  -  RIGHTS REMOVAL

  subtype rights is ordinal range 0 .. 2**20 - 1;

  no_such_right:          constant rights := 0;
  type_right_1:           constant rights := 2;
  type_right_2:           constant rights := 4;
  type_right_3:           constant rights := 8;
  delete_rights:          constant rights := 2**16;
  unchecked_copy_rights:  constant rights := 2**17;
  read_rights:            constant rights := 2**18;
  write_rights:           constant rights := 2**19;


  function Permits(
      obj:   any_access;  -- access descriptor on which rights are
                          -- checked
      r1:    rights;
      r2:    rights := no_such_right;
      r3:    rights := no_such_right;
      r4:    rights := no_such_right;
      r5:    rights := no_such_right;
      r6:    rights := no_such_right;
      r7:    rights := no_such_right)
    return boolean;

    -- Function:
    --   Checks whether the access descriptor has the specified rights.
    --   Any of r1 through r7 that are no_such_right are ignored.
    --   This operation returns true only if the access has all the
    --   rights (from 0 to 7) specified by r1 through r7.  This
    --   operation returns true if r1 through r7 are no_such_right.
    --   This operation always returns false if the access is null.
```

```
procedure Remove(
    obj:   in out any_access;  -- access descriptor on which rights
                               -- are removed
    r1:           rights;
    r2:           rights := no_such_right;
    r3:           rights := no_such_right;
    r4:           rights := no_such_right;
    r5:           rights := no_such_right;
    r6:           rights := no_such_right;
    r7:           rights := no_such_right);

  -- Function:
  --   The specified rights are removed from the access descriptor.
  --   Any of r1 through r7 that are no_such_right are ignored.
  --   All of the rights (from 0 to 7) specified are removed from the
  --   access.  A right can be removed even if it is not present on
  --   the access to begin with.  No rights are removed if the access
  --   is null.


-- TYPES  -  TYPE REPRESENTATION  -  TYPE CHECKING

-- The universal constants given below are needed for use with the
-- pragma hardware_type.  Otherwise, values of the types  base_types,
-- system_types, or processor_types are used instead of the universal
-- constants.


-- BASE TYPE REPRESENTATIONS

ds_type_rep:    constant := 0;
as_type_rep:    constant := 1;

type base_types is (ds, as);     -- data segment, access segment


-- SYSTEM TYPES AND REPRESENTATION

-- Unused system type values are reserved by Intel.

    -- DATA SEGMENT SYSTEM TYPES REPRESENTATIONS

generic_ds_type_rep:                  constant :=  0;
object_table_ds_type_rep:             constant :=  2;
instruction_ds_type_rep:              constant :=  3;
context_control_ds_type_rep:          constant :=  4;
process_control_ds_type_rep:          constant :=  5;
processor_control_ds_type_rep:        constant :=  6;
port_control_ds_type_rep:             constant :=  7;
carrier_control_ds_type_rep:          constant :=  8;
physical_storage_ds_type_rep:         constant :=  9;    -- PSO
processor_communication_ds_type_rep:  constant := 10;    -- PCO
type_control_ds_type_rep:             constant := 11;    -- TCO
refinement_control_ds_type_rep:       constant := 12;
```

```
    -- ACCESS SEGMENT SYSTEM TYPES REPRESENTATIONS

generic_as_type_rep:                      constant :=  0;
domain_as_type_rep:                       constant :=  2;
context_as_type_rep:                      constant :=  4;
process_as_type_rep:                      constant :=  5;
processor_as_type_rep:                    constant :=  6;
port_as_type_rep:                         constant :=  7;
carrier_as_type_rep:                      constant :=  8;
storage_resource_as_type_rep:             constant :=  9;
type_definition_as_type_rep:              constant := 10;


type system_types is range 0 .. 31;

    -- DATA SEGMENT SYSTEM TYPES
generic_ds:           constant system_types := generic_ds_type_rep;
object_table_ds:      constant system_types :=
                               object_table_ds_type_rep;
instruction_ds:       constant system_types := instruction_ds_type_rep;
context_control_ds:   constant system_types :=
                               context_control_ds_type_rep;
process_control_ds:   constant system_types :=
                               process_control_ds_type_rep;
processor_control_ds:constant system_types :=
                               processor_control_ds_type_rep;
port_control_ds:      constant system_types :=
                               port_control_ds_type_rep;
carrier_control_ds:   constant system_types :=
                               carrier_control_ds_type_rep;
physical_storage_ds: constant system_types :=
                               physical_storage_ds_type_rep;
processor_communication_ds:
                      constant system_types :=
                               processor_communication_ds_type_rep;
type_control_ds:      constant system_types :=
                               type_control_ds_type_rep;
refinement_control_ds:
                      constant system_types :=
                               refinement_control_ds_type_rep;

    -- ACCESS SEGMENT SYSTEM TYPES
generic_as:           constant system_types := generic_as_type_rep;
domain_as:            constant system_types := domain_as_type_rep;
context_as:           constant system_types := context_as_type_rep;
process_as:           constant system_types := process_as_type_rep;
processor_as:         constant system_types := processor_as_type_rep;
port_as:              constant system_types := port_as_type_rep;
carrier_as:           constant system_types := carrier_as_type_rep;
storage_resource_as: constant system_types :=
                               storage_resource_as_type_rep;
type_definition_as:  constant system_types :=
                               type_definition_as_type_rep;
```

```
   -- PROCESSOR TYPES AND REPRESENTATION

all_psors_type_rep:  constant := 0;
gdp_type_rep:        constant := 1;
ip_type_rep:         constant := 2;


type processor_types is (all_psors, gdp, ip);



   -- OBJECT TYPE

type object_types is
  record
    base_type:    base_types;
    sys_type:     system_types;
    psor_type:    processor_types;
  end record;


     -- DATA SEGMENT OBJECT TYPES

ds_type:                          constant object_types :=
  (ds, generic_ds, all_psors);
object_table_type:                constant object_types :=
  (ds, object_table_ds,             all_psors);
instruction_type:                 constant object_types :=
  (ds, instruction_ds,              gdp);
context_control_type:             constant object_types :=
  (ds, context_control_ds,          gdp);
process_control_type:             constant object_types :=
  (ds, process_control_ds,          gdp);
processor_control_type:           constant object_types :=
  (ds, processor_control_ds,        all_psors);
port_control_type:                constant object_types :=
  (ds, port_control_ds,             all_psors);
carrier_control_type:             constant object_types :=
  (ds, carrier_control_ds,          all_psors);
physical_storage_type:            constant object_types :=
  (ds, physical_storage_ds,         all_psors);
processor_communication_type:     constant object_types :=
  (ds, processor_communication_ds,  all_psors);
type_control_type:                constant object_types :=
  (ds, type_control_ds,             all_psors);
refinement_control_type:          constant object_types :=
  (ds, refinement_control_ds,       all_psors);
```

```
    -- ACCESS SEGMENT OBJECT TYPES

as_type:                          constant object_types :=
  (as, generic_as,                  all_psors);
domain_type:                      constant object_types :=
  (as, domain_as,                   all_psors);
context_type:                     constant object_types :=
  (as, context_as,                  gdp);
process_type:                     constant object_types :=
  (as, process_as,                  gdp);
processor_type:                   constant object_types :=
  (as, processor_as,                gdp);
port_type:                        constant object_types :=
  (as, port_as,                     all_psors);
carrier_type:                     constant object_types :=
  (as, carrier_as,                  all_psors);
storage_resource_type:            constant object_types :=
  (as, storage_resource_as,         all_psors);
type_definition_type:             constant object_types :=
  (as, type_definition_as,          all_psors);


    -- CHECKING

function Check_base_type(
    obj:  any_access; -- access descriptor for the object to be
                      -- checked
    bt:   base_types) -- base_type to check against
  return boolean;

  -- Function:
  --   If obj is null, then return true.  Otherwise, if the referenced
  --   object descriptor is not a storage or refinement descriptor,
  --   then return false.  Otherwise, return true if and only if the
  --   referenced object has the specified base type.


function Check_system_type(
    obj:  any_access;   -- access descriptor for the object to be
                        -- checked
    t:    object_types) -- object_type containing base type and
                        -- system type to check against
  return boolean;

  -- Function:
  --   If obj is null, then return true.  Otherwise, if the referenced
  --   object descriptor is not a storage or refinement descriptor,
  --   then return false.  Otherwise, return true if and only if the
  --   referenced object has the specified base type and system type
  --   (the processor type field is ignored).
```

```
function Check_object_type(
    obj:  any_access;    -- access descriptor for object to be checked
    t:    object_types) -- object_type to check against
  return boolean;

  -- Function:
  --    If obj is null, then return true.  Otherwise, if the referenced
  --    object descriptor is not a storage or refinement descriptor,
  --    then return false.  Otherwise, return true if and only if the
  --    referenced object has the specified object type (base type,
  --    system type, and processor class).


function Check_extended_type(
    obj:  any_access)  -- access descriptor for object to be checked
  return boolean;

  -- Function:
  --    If obj is null, then return true.  Otherwise, return true if
  --    and only if the referenced object is an instance of an
  --    extended type (i.e., the referenced object descriptor is a
  --    type descriptor).


-- MEMORY RELATED DEFINITIONS

max_physical_mem_sz: constant := 2**24;  -- up to 16 Mbytes in
                                         -- physical memory

subtype seg_base_adr is ordinal range 0 .. max_physical_mem_sz - 1;

max_seg_sz: constant := 2**16;  -- up to 65,536 bytes in one segment


-- OBJECT TABLE DEFINITIONS

type objtab_index is range 0 .. 2**12 - 1;  -- up to 4,096 object
                                            -- table entries in an
                                            -- OT (including header)

nil_objtab_index: constant objtab_index := 0;
```

```
-- TYPE RIGHTS DEFINITIONS

type packed_boolean is new boolean;

subtype type_rights_index is short_ordinal range 0 .. 2;

type type_rights is array (type_rights_index) of packed_boolean;


-- ACCESS DESCRIPTOR DEFINITIONS
type ac_descr is
  record
    valid:              boolean;      -- if false, access descriptor
                                      -- is null
    type_rts:           type_rights;  -- meaning depends on the
                                      -- system type
    OT_index:           objtab_index; -- index of object descriptor
                                      -- for referenced object in
                                      -- object table
    del_rts:            boolean;      -- if true, this AD can be
                                      -- overwritten
    unchecked_copy_rts: boolean;      -- if true, no level check is
                                      -- done when this AD is copied
    read_rts:           boolean;      -- if true, this AD can be used
                                      -- to read the object
    write_rts:          boolean;      -- if true, this AD can be used
                                      -- to write the object
    dir_index:          objtab_index; -- index of object descriptor
                                      -- for object table in object
                                      -- table directory
  end record;

ac_descr_sz: constant := 4;
  -- size of an access descriptor in bytes.

null_ad: constant ac_descr := ac_descr'(
              valid                 => false,
              type_rts          => type_rights'(false, false, false),
              OT_index              => nil_objtab_index,
              del_rts               => false,
              unchecked_copy_rts    => false,
              read_rts              => false,
              write_rts             => false,
              dir_index             => nil_objtab_index);

function Inspect_access(
    a:  any_access)   -- access descriptor to be inspected
  return ac_descr;

  -- Function:
  --   Return the access descriptor represented as an ac_descr record.
  --   The record representation can neither be used as an access
  --   nor converted into one (not even using Unchecked_conversion).
-- OBJECT TABLE ENTRIES

OT_entry_sz:      constant := 16;
  -- size of an object descriptor in bytes.

type OT_entry_type is (non_allocatable, extended, refinement,
                       storage);

type OT_entry_subtype is (header, free, invalid_interconnect,
                          interconnect, dont_care);
    -- dont_care is needed for compiler in specifying subtypes.

type non_allocatable_descr(entry_type:  OT_entry_type :=
                                          non_allocatable;
                           entry_subtype:  OT_entry_subtype :=
                                             header)
is
  record
    case entry_subtype is
      when interconnect =>
        base_adr:     seg_base_adr;
          -- physical address in interconnect address space.
        length:       short_ordinal;
          -- length-1 of interconnect segment.
      when free =>
        next_free:   objtab_index;    -- link in free entry list.
      when header =>
        free_index:  objtab_index;    -- if heap, index of 1st free
                                      -- entry in list.
                                      -- if stack, index of last
                                      -- allocated entry.
        end_index:   objtab_index;    -- if heap, then 0.
                                      -- if stack, index of last entry
                                      -- in table.
        fault_level: short_ordinal;   -- if heap, then 0.
                                      -- if stack, level number for
                                      -- return fault.
                                      -- (0 if no return fault is set)
        cur_level:   short_ordinal;   -- level number used to init new
                                      -- descrs.
        stor_claim:  integer;         -- if heap, then storage claim.
                                      -- (Can be infinite storage
                                      -- claim).
                                      -- if stack, then =
                                      -- infinite_storage_claim.
      when invalid_interconnect | dont_care => null;
    end case;
  end record;
```

```
type OT_entry (entry_type:    OT_entry_type := non_allocatable) is
  record
    case entry_type is
      when non_allocatable =>
        entry_subtype: OT_entry_subtype;
          -- Ada won't allow more than one (not nested) variant part
          -- so the description of non_allocatable object_descr's
          -- requires a separate record structure and the use of
          -- retypes or Unchecked_conversion to get the types right.
      when others =>    -- refinement, storage, type descriptors
        valid:             boolean;  -- descr. validity
        copied:            boolean;  -- gray bit for garbage
                                     -- collection.
        level:             short_ordinal;  -- object level number.
        case entry_type is
          when extended =>
            privat:        boolean;  -- if true, then private, if
                                     -- false, then public.
                                     -- e is omitted to avoid
                                     -- conflict with reserved word
            twhite:        boolean;  -- if false, then black (used by
                                     -- garbage collection)
            type_def_ref:  ac_descr; -- reference to type definition
                                     -- object.
            typed_obj_ref: ac_descr; -- reference to typed object
          when refinement | storage =>
            base_type:     base_types;  -- ds (data segment) or as
                                        -- (access segment)
            sys_type:      system_types;
            psor_type:     processor_types;
            length:        short_ordinal; -- length-1 of segment.
            case entry_type is
              when refinement =>
                byp_ot_index:   objtab_index;
                                     -- object coordinates of
                                     -- refined storage descriptor.
                byp_dir_index:  objtab_index;
                rwhite:         boolean;  -- if false, then black.
                                          -- (used by garbage
                                          -- collection)
                base_disp:      short_ordinal;
                                     -- byte displacement of base
                                     -- of refinement in underlying
                                     -- storage segment
                rfn_obj_ref:    ac_descr;
                                     -- reference to object for which
                                     -- this is a refinement.
              when storage =>
                allocated: boolean;  -- if true then base address is
                                     -- valid
                windowed:  boolean;  -- if true, then qualified in IP
                                     -- window.
                altered:   boolean;  -- set when segment written
                accessed:  boolean;  -- set when segment read or
                                     -- written
                base_adr:  seg_base_adr; -- physical base address of
                                     -- segment.
                dirty:     boolean;  -- if true, then segment contains
                                     -- some non-zero bits.
                swhite:    boolean;  -- if false, then black. (used by
                                     -- garbage collection)
                prv_level: short_ordinal;  -- (used only for contexts)
                                     -- previous level is level of
                                     -- current context minus one
              when others => od_dummy1: byte;
            end case;
          when others => od_dummy2: byte;
        end case;
    end case;
  end record;


subtype storage_descr is OT_entry(entry_type => storage);

subtype refinement_descr is OT_entry(entry_type => refinement);

subtype type_descr is OT_entry(entry_type => extended);


subtype free_descr is non_allocatable_descr(entry_type =>
  non_allocatable, entry_subtype => free);

subtype interconnect_descr is non_allocatable_descr(entry_type =>
  non_allocatable, entry_subtype => interconnect);

subtype invalid_interconnect_descr is
                                non_allocatable_descr(entry_type =>
  non_allocatable, entry_subtype => invalid_interconnect);

subtype object_table_header is non_allocatable_descr(entry_type =>
  non_allocatable, entry_subtype => header);
```

```
-- INSPECT OBJECT OUTPUT

type inspect_object_output (valid: boolean := true) is

  -- This type is returned by the hardware Inspect Object instruction.
  -- It consists of the record representation of an access descriptor
  -- and in case the access desciptor is valid, the record
  -- representation of the object descriptor it references.

  record
    type_rts:    type_rights;    -- meaning depends on the system type
    OT_index:    objtab_index;   -- index of object descriptor for
                                 -- referenced object in object table
    del_rts:     boolean;   -- if true, this AD can be overwritten
    unchecked_copy_rts:
                 boolean;   -- if true, no level check is done when this
                            -- AD is copied
    read_rts:    boolean;   -- if true, this AD can be used to read the
                            -- object
    write_rts:   boolean;   -- if true, this AD can be used to write
                            -- the object
    dir_index:   objtab_index;
                            -- index of object descriptor for object
                            -- table in object table directory
    case valid is
      when false => dummy_od: byte;
      when true  => od:       OT_entry;
    end case;
  end record;


function Inspect_object(
    obj:  any_access)  -- access descriptor for object to be inspected
  return inspect_object_output;

  -- Function:
  --    Returns a record representation of an
  --    access descriptor, and, in case the AD is valid (not null),
  --    a record representation of the object descriptor it references.
  --    The record representation of the AD can neither be used as an
  --    access nor converted into one (not even using
  --    Unchecked_conversion).


end Descriptor_Definitions;
```

## IMAX DEFINITIONS PACKAGE

```
with Unchecked_conversion, Descriptor_Definitions;
package iMAX_Definitions is

  -- Function:
  --   This package contains miscellaneous definitions needed by
  --   iMAX users.


  use Descriptor_Definitions;


  --  ACCESS SUBTYPES CORRESPONDING TO HARDWARE OBJECT TYPES

      -- ANY ACCESS SEGMENT  -  ANY DATA SEGMENT
  subtype any_as is any_access;
  subtype any_ds is any_access;

      -- ACCESS SEGMENT TYPES
  subtype access_sgt        is any_access;
  subtype domain            is any_access;
  subtype context           is any_access;
  subtype type_definition   is any_access;
  subtype process           is any_access;
  subtype port              is any_access;
  subtype carrier           is any_access;
  subtype processor         is any_access;
  subtype storage_resource  is any_access;

      -- DATA SEGMENT TYPES
  subtype data_sgt                  is any_access;
  subtype object_table              is any_access;
  subtype instruction_segment       is any_access;
  subtype context_control           is any_access;
  subtype process_control           is any_access;
  subtype port_control              is any_access;
  subtype carrier_control           is any_access;
  subtype processor_control         is any_access;
  subtype processor_communication   is any_access;
  subtype physical_storage          is any_access;
  subtype type_control              is any_access;
  subtype refinement_control        is any_access;

      -- EXTENDED TYPE
  subtype extended_type is any_access;
```

```
-- PROCEDURE VALUE

package type procedure_val is
  --
  -- Function:
  --   Introduced to specify parameterless procedures as parameters
  --   to procedures.

  procedure Main;

end procedure_val;



-- DEFAULT PROCESS TUNING PARAMETERS

type deadline_scheduling_value is new short_integer
                                        range -(2**14) .. (2**14) - 1;
  -- more positive values are more urgent

default_service_period: constant short_ordinal := 400;
  -- about 0.1 seconds
default_deadline:       constant deadline_scheduling_value := 1600;
  -- about 0.4 seconds
default_priority:       constant short_ordinal := 10;



-- ANY ACCESS ARRAY

-- These definitions are useful to establish addressability to  an
-- access segment.

max_access_array_sz:  constant := max_seg_sz / ac_descr_sz;
    -- maximum number of access descriptors in an access list

subtype access_array_index is short_ordinal
                                range 0 .. max_access_array_sz - 1;

type any_access_array_val is array (access_array_index) of any_access;

type any_access_array is access any_access_array_val;

function Retype_to_any_access_array
  -- ( a:    any_as)
  --  return any_access_array;
  is new Unchecked_conversion(any_as, any_access_array);

  -- Function:
  --   The given access descriptor for an access segment is retyped
  --   into any_access_array.  This operation does not change the
  --   rights of the given access descriptor.
-- BYTE ARRAY

-- These definitions are useful to establish adressability to a data
-- segment.

type byte_array_val is array (short_ordinal range 0 .. max_seg_sz - 1)
                         of byte;

type byte_array is access byte_array_val;

function Retype_to_byte_array
  -- ( a:    any_ds)
  --  return byte_array;
  is new Unchecked_conversion(any_ds, byte_array);

  -- Function:
  --   The given access descriptor for a data segment is retyped into
  --   byte_array.  This operation does not change the rights
  --   of the given access descriptor.


-- ACCESS SELECTOR

subtype EAS_selector is short_ordinal range 0 .. 3;

type access_selector is
  record
    EAS:    EAS_selector;
    index:  access_array_index;
  end record;

null_access_selector:  constant access_selector :=
                                 access_selector'(0, 0);


-- IDLE

procedure Idle(
    time:  short_ordinal range 0 .. 2 ** 14 - 1);  -- idling time

  -- Function:
  --   The calling process is delayed for the specified number of
  --   system time units.  If the time parameter is zero, the
  --   procedure returns immediately.  One system time unit is
  --   typically 200 microseconds.
  --!! A system time unit is a hardware constant, 256 microseconds
  --!! for 432/600 systems.
     Note:
  --   This is an inline procedure, so that the time needed to
  --   execute call and return need not be considered.
  -- MISCELLANEOUS DEFINITIONS

  type print_name is array (1 .. 15) of character;

end iMAX_Definitions;
```

## BASIC DEFINITIONS EXAMPLES PACKAGE

```
with iMAX_Definitions, Descriptor_Definitions;
package Basic_Definitions_Examples is

  -- Function:
  --   Give examples of using basic definitions:
  --     1) Define type-specific rights.
  --     2) Function that returns length of segment
  --     3) Function that returns level number of object
  --     4) Function that does a level check and checks delete rights
  --        to determine whether one AD can be copied over another.

  use iMAX_Definitions, Descriptor_Definitions;


  -- an example of defining type-specific rights
  -- (for instruction segments).

  call_rights:  constant rights := type_right_1;
  trace_rights: constant rights := type_right_2;


  Function Length(
      obj: any_access)
    return ordinal;

    -- Function:
    --   Return the length of the referenced segment in bytes.  If the
    --   access is null or references an extended-type object,
    --   return zero.


  Function Level(
      obj: any_access)
    return short_ordinal;

    -- Function:
    --   Return the level number of the referenced object.

    -- Exceptions:
    --   constraint_error -- raised if obj is null
  Function Copy_AD_OK(
      src:   any_access;            -- source AD
      dseg:  any_access_array;      -- AD for segment containing dest. AD
      dslot: access_array_index) -- index of destination AD slot
    return boolean;

    -- Function:
    --    Return true if the destination AD can be overwritten and if
    --    a copy from the source AD to the destination AD slot passes
    --    level checking.

end Basic_Definitions_Examples;
```

## BASIC DEFINITIONS EXAMPLE PACKAGE BODY

```
with iMAX_Definitions, Descriptor_Definitions;
package body Basic_Definitions_Examples is

  use iMAX_Definitions, Descriptor_Definitions;

  function Length(
      obj: any_access)
    return ordinal is

      descr_image: OT_entry;

    begin
      if obj = null then
        RETURN 0;

      else
        descr_image := Inspect_object(obj).od;
        case descr_image.entry_type is
          when extended =>
            RETURN 0;

          when refinement |
               storage =>
            RETURN ordinal(descr_image.length)+1;

          when non_allocatable =>
            case descr_image.entry_subtype is
              when interconnect =>
                RETURN ordinal(descr_image.length)+1;

              when others =>
                null;

            end case;
        end case;
      end if;
  end Length;
  function Level(
      obj: any_access)
    return short_ordinal is

    descr_image: OT_entry;

  begin
    if obj = null then
      RAISE constraint_error;

    else
      RETURN Inspect_object(obj).od.level;

    end if;
  end Level;


  function Copy_AD_OK(
      src:   any_access;            -- source AD
      dseg:  any_access_array;      -- AD for access segment containing
                                    -- destination AD
      dslot: access_array_index)    -- index of destination AD slot
    return boolean is

  begin

    return
      (dseg(dslot)=null or else Permits(dseg(dslot), delete_rights))
      and then
      (src=null or else Level(src)<=Level(any_access(dseg)));

  end Copy_AD_OK;

end Basic_Definitions_Examples;
```

# CHAPTER STO
# STORAGE MANAGEMENT

This chapter describes how iMAX manages storage using 432 storage resource objects (SROs). Users should read this chapter to understand the unique storage management services provided by iMAX even if they do not need to use the visible interfaces to storage management.

The services described in this chapter support the Ada concepts of access types and allocators (the Ada new operator). Using Ada, you can dynamically create objects with a range of lifetimes. Storage reclamation is provided by iMAX and the 432 hardware. Users need to explicitly reference the packages described in this chapter only if they have special requirements outside of normal applications programming needs, such as:

- a need to explicitly control the structuring of an object into access segments and data segments in the most straightforward way.

- a need for some or all of their program to reside in "frozen" memory (memory which is not subject to relocation of segments and relocation faults).

## V2 CAPABILITIES

iMAX V2 provides a real-memory system with:

- dynamic allocation of objects

- transparent expansion of object tables and stack or heap storage blocks, as required by user processes

- storage reclamation transparent to users

- a range of lifetimes for created objects

iMAX V2 does not support virtual memory. iMAX V2 also does not support limits on the amount of memory used by a particular process or collection of dynamically-allocated objects. These capabilities will be provided in future iMAX versions.

## STORAGE RESOURCE OBJECTS

Creating a new 432 object involves the allocation of two types of resources:

- a new object descriptor for the new object -- this is allocation of the virtual address space of the 432
- a continuous block of physical storage for the new object -- this is allocation of the physical address space of the 432

Note that the creation of some objects, such as refinements and extended-type objects, requires a new object descriptor but no additional physical storage.

A storage resource object provides access to both free object descriptors and blocks of free physical memory.

If either of these resources is exhausted (no free object table entries or no storage block large enough) when an object creation is attempted, then iMAX allocates more resources transparently to the user. If necessary, the user process is blocked until the needed resources are available.

Users view the free storage in an iMAX system as a collection of SROs, each representing an unbounded claim on free memory, and each specifying the storage management attributes of objects created from it. Each SRO specifies lifetime strategy and memory type for objects allocated from it. These attributes of an SRO or an object cannot be changed.

## LIFETIME STRATEGIES

A basic characteristic of storage management in both Ada and iMAX is that storage reclamation is transparent. User programs create objects but do not need to delete them -- iMAX detects when objects are no longer used and reclaims them. An object's lifetime strategy determines when and how it is deallocated, and is derived from the lifetime strategy of the SRO used to create the segment.

### STACK LIFETIMES

The most restrictive (and most efficient to deallocate) lifetime strategy restricts access to objects to the context that creates them and to subordinate contexts. The 432 hardware automatically deallocates such objects on returning from the context that creates them. This is the stack lifetime strategy.

Access descriptors for stack-allocated objects are confined to objects with the same or shorter lifetimes (i.e., objects with lifetimes tied to the same context or to a subordinate context). For example, a context cannot return an AD for a stack-allocated object to its caller. On the other hand, passing an AD for a stack-allocated object as a parameter to a subordinate context is always allowed, because the called context always has a shorter lifetime than the calling context.

Each process has an associated stack SRO. These SROs are bound to their associated processes; stack SROs cannot be created or referenced as objects distinct from processes. The process stack SRO is implicitly used by the hardware CALL and RETURN instructions. CALL creates a context object (a subprogram activation record) from the executing process's stack SRO. The stack SRO is also used to create objects local to the context that are created by the context. RETURN deletes both the context object and objects local to the context that are created by the context.

## GLOBAL HEAP LIFETIMES

The least restrictive (and least efficient to deallocate) lifetime strategy is to do an exhaustive search of memory that determines what objects are no longer reachable from the programs in the system via a chain of ADs. Since the only way that an object can be used by a program is via an AD, an object that cannot be reached from any program is unusable and can be deleted. Such unreachable/unusable objects are called garbage, and the iMAX program that finds and reclaims garbage objects is called the garbage collector. The iMAX garbage collector executes as a separate process in an iMAX system. The garbage collector process is able to run concurrently with other iMAX and user processes, and its operation does not interfere with the operation of any other processes in the system. The lifetime strategy which reclaims objects only via garbage collection is the global heap lifetime strategy. Objects allocated from a global heap SRO have lifetimes that are not limited by the process or context in which they are created.

A V2 system has two global heap SROs, one to allocate frozen memory (non-relocatable) and the other to allocate normal memory (relocatable). The Memory_Controller package provides access to the two global heap SROs. Also, each user process has an access for one of these global heap SROs, via the default_global_heap_SRO field of its process globals object. This default SRO access is used by the Ada compiler, and also by some iMAX operations when an SRO parameter is defaulted. The default_global_heap_SRO associated with a process is determined when the process is created and cannot be changed later. The process globals object and how to reference it are described in Chapter PEN, Program Environment Access.

## LOCAL HEAP LIFETIMES

The third strategy is a hybrid of the other two: a heap SRO that is local to a context. Objects created from such a local heap SRO are deleted in one of two ways. First, during the life of the associated context, the system-wide garbage collection process reclaims unreferenced objects found in the local heap. Second, on returning from the context, the local heap and all objects allocated from it and not previously garbage collected are deleted.

The SRO_Manager operation Create_local_heap creates a local heap SRO and ties its lifetime and the lifetime of objects created from it to the lifetime of a specific context object. This context object is specified by giving its position relative to the current context in the chain of active contexts associated with the executing process. For example, suppose A calls B calls C which calls Create_local_heap. If 0 is passed, then the local heap's lifetime is tied to C's context object, if 1 to B's, etc. Now suppose 1 is passed; a local heap SRO is created that is tied to the lifetime of B's context. C, and any other procedure that C calls, can use this SRO to create objects that have the same lifetime (and confinement) as objects created by B using the process stack SRO. When B returns, the local heap and objects created from it are deleted.

## STORAGE MANAGEMENT TRANSITIONS

Figure STO-1 ties together the three lifetime strategies by showing the transitions of storage between free memory, allocated objects, and garbage for all three strategies.

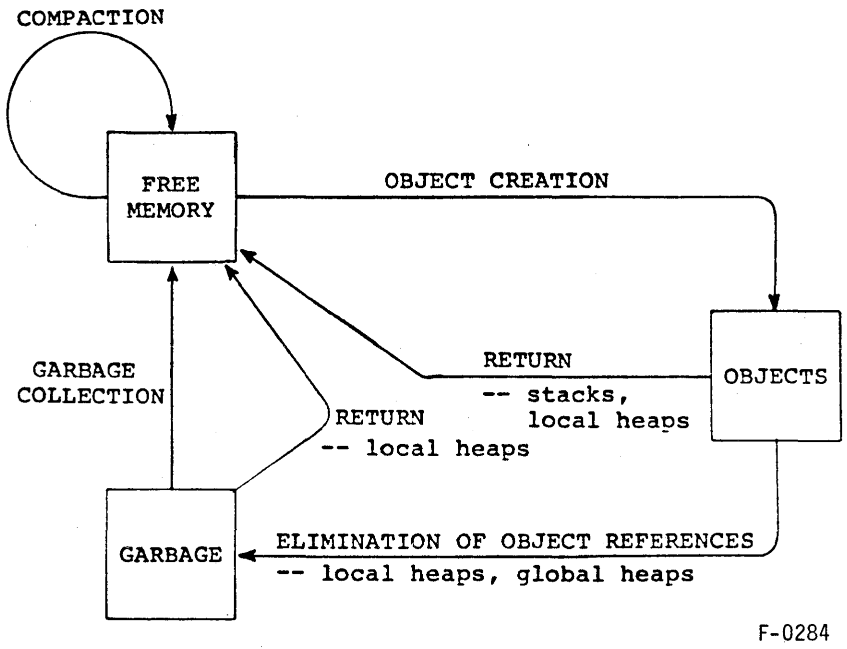


Figure STO-1. Storage Management Transitions

## FRAGMENTATION AND COMPACTION

Fragmentation is the division of free physical storage into non-contiguous blocks as the result of allocations and deallocations. Due to fragmentation, a segment allocation request can fail even if the total amount of free storage is larger than the amount requested, because no contiguous block is sufficiently large.

Compaction reduces fragmentation by relocating objects in physical memory to reduce the number and increase the size of the free storage blocks. Compaction increases the quality (in larger block size and reduced number of blocks to search) of free storage. Garbage collection, in contrast, increases the quantity of free storage. iMAX compaction runs as needed as an asynchronous process, invisible to users. Compaction can cause delays in user processing by relocating objects (relocating an object makes it temporarily inaccessible and any reference to an object being relocated delays the process making the reference). Compaction also blocks certain other iMAX services, which can delay users of those services even if the user programs are not relocated themselves.

## MEMORY TYPE

Memory type is an attribute of SROs (and of all segments) provided by iMAX to distinguish memory that is subject to segment relocation from memory that isn't. Relocation can temporarily make segments inaccessible, resulting in faults or delays. Some parts of iMAX and possibly some user applications cannot tolerate such faults or delays and require memory that is never relocated.

iMAX divides the physical memory of a 432 system into frozen and normal memory. Segments in normal memory can be relocated; segments in frozen memory cannot be. Garbage collection applies to both types of memory. All frozen memory, whether allocated or not, is in one contiguous part of memory. All normal memory, whether allocated or not, is in another contiguous region. There is a shifting boundary between the two regions, set to minimize the amount of frozen memory. iMAX V2 provides one global heap SRO for frozen memory and one for normal memory.

Users concerned with time-critical processing can require parts of their programs to run in frozen memory in order to avoid segment validity faults. (These faults can occur when segments are relocated in normal memory.) Because such faults are invisible to the user program, rarely occur, and result in little performance degradation, frozen memory need only be used in the presence of stringent time requirements. It must be realized that because frozen memory cannot be compacted, frequent allocations and deallocations of frozen memory may result in severe and irreparable fragmentation of the frozen part of physical memory.

## RUNNING OUT OF MEMORY

When a storage request cannot be satisfied, the requesting process is enqueued in a list of processes waiting for more storage. There are two such lists, for processes waiting for frozen and normal memory respectively. When garbage collection, compaction, or the actions of still-running processes make enough free storage available, then outstanding requests are satisfied and processes are removed from the waiting lists, able to run again. However, if a process makes a storage request that cannot ever be satisfied, it simply waits indefinitely. There is no way for an iMAX user to detect when a user process is permanently blocked because it is waiting for memory. Another user process can use iMAX's Basic_Process_Management services to destroy such a blocked process and make more storage available for other uses.

Ada defines the storage_error exception, to be raised when the dynamic storage claim allocated to a task (process) or to a collection of dynamic objects is exhausted. However, limits on storage claims are not implemented in iMAX V2, and storage_error is never raised.

## PACKAGE DESCRIPTIONS

The user interface to storage management consists of two packages, SRO Manager and Memory Controller. The SRO_Manager package manages storage resource objects. It contains operations that:

- create a new instance of one SRO type, local heap SROs.
- correspond to the hardware instructions for creating generic segments and refinements.

iMAX does not support creation of new global heap SROs. A new stack SRO is created for each new process. (Creating processes is described in Chapter BPM, Basic Process Management.)

SRO_Manager provides six procedures that correspond to the three 432 instructions that create generic objects: CREATE ACCESS SEGMENT, CREATE DATA SEGMENT, and CREATE REFINEMENT. Two Ada procedures correspond to each instruction, the first to create the object from a heap SRO and the second to create the object from the process stack SRO. For all six procedures, the length of the new object is specified by a short_ordinal that is actually size in bytes minus one. Therefore, the size of the new segment or refinement must be in the range 1 .. 65536 bytes. To create an access segment with n ADs, specify a length parameter value of (4*n)-1.

Newly created data segments are initially all zeroes. Newly created access segments are initially all null access values.

If the SRO access parameter is omitted or null in calling create operations that use a heap SRO, then the default_global_heap_SRO referenced by the calling process is used.

The access returned for the new object by any of these procedures has all access rights.

SRO_Manager also defines the hardware-recognized create rights on SRO accesses. Create rights are required on any SRO access used for a Create operation, even if the operation is defined in some other iMAX package (e.g., Untyped_Ports. Create_carrier). If an SRO access without create rights is used for a Create operation, then the Ada exception constraint_error is raised.

The Memory_Controller package provides accesses for the two global heap storage resources in V2 (one frozen and one normal), and also a function to return memory usage information. The global heap SRO accesses both have create rights.

## EXAMPLES

No examples of using SRO_Manager or Memory_Controller are given in this chapter. The Stack_Manager package body in Chapter EXT, Extended Types, is an example of how to use the SRO_Manager package.

## SRO MANAGER PACKAGE

```
with Descriptor_Definitions, iMAX_Definitions;
package SRO_Manager is

  -- Function:
  --   This package provides a low-level interface to memory management.
  --   All but the Create_local_heap function are implemented as
  --   432 instructions.  The "heap" allocation instructions take an
  --   optional parameter, i.e. a default SRO.  At compile time this
  --   parameter defaults to null, but at run time will default to
  --   the default local heap SRO in the process globals object of
  --   the executing process.  The "stack" allocation instructions
  --   do not need an SRO parameter since the stack SRO is referenced
  --   implicitly.

  --   In V2, only one system right is interpreted for SRO's, i.e.
  --   create rights.  This right is required for all create
  --   operations.


  use iMAX_Definitions;

  -- System Rights for SRO access descriptors:

  create_rights: constant Descriptor_Definitions.rights :=
                            Descriptor_Definitions.type_right_1;


  procedure Create_data_segment(
      length:     short_ordinal;  -- length of segment - 1;
      dseg:   out data_sgt;       -- data segment created.
      SRO:        storage_resource := null);
                                  -- SRO for create.

    -- Function:
    --   A heap data segment of the specified size is created.
    --   If the SRO parameter is defaulted, then the default local
    --   heap SRO in process globals is used for the create.


  procedure Create_access_segment(
      length:     short_ordinal;  -- length of segment (in bytes) - 1;
      aseg:   out access_sgt;     -- access segment created.
      SRO:        storage_resource := null);
                                  -- SRO for create.

    -- Function:
    --   A heap access segment of the specified size is created.
    --   If the SRO parameter is defaulted, then the default local
    --   heap SRO in process globals is used for the create.
```

```
procedure Create_generic_refinement(
    obj:          any_access;       -- object to be refined
    offset:       short_ordinal;    -- offset of the refinement in bytes
    length:       short_ordinal;    -- length of the refinement minus one
                                    -- in bytes
    rfn:      out any_access;       -- the resulting refinement.
    sro:          storage_resource := null);
                                    -- SRO for create.

  -- Function:
  --   A heap refinement is created from the specified object,
  --   beginning at offset (bytes) and length (bytes) long. The base
  --   type of the created refinement will be the same as the base
  --   type of the original object.  Its system type will be generic.


procedure Create_stack_data_segment(
    length:      short_ordinal;  -- length of segment - 1;
    dseg:    out data_sgt);      -- data segment created.

  -- Function:
  --   A stack data segment of the specified size is created.


procedure Create_stack_access_segment(
    length:      short_ordinal;  -- length of segment (in bytes) - 1;
    aseg:    out access_sgt);    -- access segment created.

  -- Function:
  --   A stack access segment of the specified size is created.


procedure Create_stack_generic_refinement(
    obj:          any_access;       -- object to be refined
    offset:       short_ordinal;    -- offset of the refinement in bytes
    length:       short_ordinal;    -- length of the refinement minus one
                                    -- in bytes
    rfn:      out any_access);      -- the resulting refinement.
                                    -- SRO for create.

  -- Function:
  --   A stack refinement is created from the specified object,
  --   beginning at offset (bytes) and length (bytes) long. The base
  --   type of the created refinement will be the same as the base
  --   type of the original object.  Its system type will be generic.
  level_error: exception;  -- Can be raised by Create_local_heap.

  function Create_local_heap(
      relative_level:      short_ordinal := 0) -- relative level number.
    return storage_resource;

    -- Function:
    --   This function creates a local heap SRO.
    --   The parameter determines the lifetime of the local heap SRO
    --   and the objects created from it.  Specifically, the parameter
    --   selects a context object (i.e. its lifetime).  For example,
    --   if A calls B calls C which calls this function passing 0,
    --   the lifetime of the created local heap SRO will be tied
    --   to C's context; if 1 is passed, B's context; if 2, C's, etc.
    --   If the parameter exceeds the depth of the process's Context
    --   stack, a "level_error" exception is raised.

end SRO_Manager;
```

MEMORY CONTROLLER PACKAGE

```
with iMAX_Definitions;
package Memory_Controller is

  -- Function:
  --   This package should only be used by a high-level process/memory
  --   manager and not by a general user.  This package specification
  --   will be different in version 3 of iMAX.  Hence, a use of this
  --   package by some program will prevent that program from being
  --   compatible with iMAX V3.


  type primary_memory_info is
    record
      memory_size:                ordinal;
      normal_memory_size:         ordinal;
      normal_free:                ordinal;
      frozen_free:                ordinal;
      process_waiting_for_normal: boolean;
      process_waiting_for_frozen: boolean;
    --
    -- "frozen_memory_size" equals memory_size - normal_memory_size.
    -- "normal_allocated" equals normal_memory_size - normal_free.
    -- "frozen_allocated" equals frozen_memory_size - frozen_free.
    -- the "process_waiting" flags indicate that one or more processes
    -- are waiting for an event list for memory.
    end record;


  -- In version 2, the "real memory" system, there are only two
  -- global heap SROs, one for normal memory and one for frozen.
  --
  normal_global_heap_sro:    iMAX_Definitions.storage_resource;

  frozen_global_heap_sro:    iMAX_Definitions.storage_resource;


  function Get_primary_memory_data
    return primary_memory_info;

    -- Function:
    --   This function returns information about primary memory.
    --   This information is obtained by sampling the memory management
    --   data structures without locking them.  Hence, it is possible
    --   for inconsistent information to be obtained.

end Memory_Controller;
```

# CHAPTER BPM
# BASIC PROCESS MANAGEMENT

This chapter describes how iMAX controls and organizes collections of concurrent processes. iMAX extends the processor-interpreted process object to include additional state and structure information and to support additional operations. iMAX process management is especially important because of the 432's unique ability to execute multiple processes simultaneously and transparently on multiple processors.

The primary purpose of the Basic_Process_Management package (BPM) described in this chapter is to allow Intel and its customers to easily build higher-level process managers that include resource control and more sophisticated scheduling. BPM provides an essentially unprotected interface to processes and their scheduling, while guaranteeing the integrity of the system objects used by the hardware.

iMAX also lets users create static processes at system initialization which cannot be controlled using BPM. Static processes are described in Chapter CON, Configuration.

Users of BPM take responsibility for setting their process scheduling parameters so that they do not overcommit the system's processing power. It is possible for the user to set those parameters so that they lock out normal iMAX function (Appendix PRS, Process Scheduling Information, gives scheduling information for those V2 processes that compete with user processes to be dispatched). Also, BPM provides little control over the ways in which incorrect processes can interfere with other processes.

iMAX process management is distinct from Ada's tasking facilities. Table BPM-1 compares Ada tasking with iMAX basic process management, and should be helpful to system designers who must choose one or the other (or some combination) for a particular application. Note that the Intel 432 Ada compiler does not yet support tasking. This table becomes more useful when the compiler does support tasking.

Table BPM-1. Comparison of Ada Tasks and iMAX BPM Processes

| Attribute | Ada tasks | iMAX BPM processes |
|---|---|---|
| Scheduling | Advisory priorities fixed at compile-time | User can dynamically vary: priorities, deadlines, time slice length, and number of time slices before rescheduling consideration |
| Hierarchy | Implied by nesting of declarations. When a task is aborted, any dependent tasks are aborted. | Processes can be organized into trees and process operations can apply to entire trees |
| Control | Abort another task, raise Failure exception in another task | Start, Stop, Reset, Restart, and Destroy other processes. Use guardian ports to receive and restart processes suspended by some condition. |
| Communication | A task can wait for multiple entries guarded by conditions. Timeouts can be included in the alternatives waited for. | Processes send or receive messages via explicitly-identified ports. Surrogate operations support prioritized queuing of messages and waiting for the occurrence of one of several different events. |
| Mutual Exclusion | Not explicitly provided, but can be constructed using communication facilities. | Not explicitly provided, but can be constructed using communication facilities. |
| Portability | Ada tasks are part of standard Ada. | iMAX processes are not part of standard Ada. |

## PROCESS TREES

iMAX extends the hardware definition of processes to support trees of processes. These process trees are useful because iMAX provides operations that manipulate groups of processes in a tree as if they are a single process (e.g., start, stop, or destroy an entire process tree). This hierarchial capacity allows the design of software to exercise control over a computation without knowing or caring if it is structured as a single process or contains a dozen subordinate processes.

For example, suppose that, in a data collection system, process MAIN activates independent process FFT to perform a computation called a "fast Fourier transform" on a large data set. Internal to FFT and unknown to MAIN, the FFT process is implemented with two dependent subprocesses, FFT1 and FFT2, which each operate on half of the data set. Each of these processes may also be implemented with two dependent subprocesses, which each operates on half of the half data set given to its parent. The parent processes simply spawn the subprocesses, activate them, wait for their successful completion, and then merge their results. With this divide and conquer strategy and a multiple processor system, a large computation can be finished in much less time than it would otherwise take, and can still be controlled by MAIN as if it were a single process. Examples of the control that MAIN might exercise are: (1) destroying the process tree rooted in FFT if a user requests that the command be aborted. (2) stopping the process tree rooted in FFT to free up resources (processing time) for a more important activity, and then restarting it.

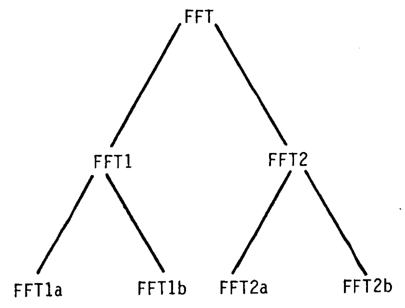


Figure BPM-1. Process Tree Example

Each process's position in the tree structure is fixed when the process is created and is determined by the heap SRO used to create it. If a global heap SRO is used, then the new process is the root of a new process tree, and is subordinate to no other process. If a local heap SRO is used to create a process, then the new process is a "child process" of the process that created the local heap SRO. Chapter STO, Storage Management, describes how to create a local heap which can be used to create child processes. Figure BPM-2 illustrates process tree creation.

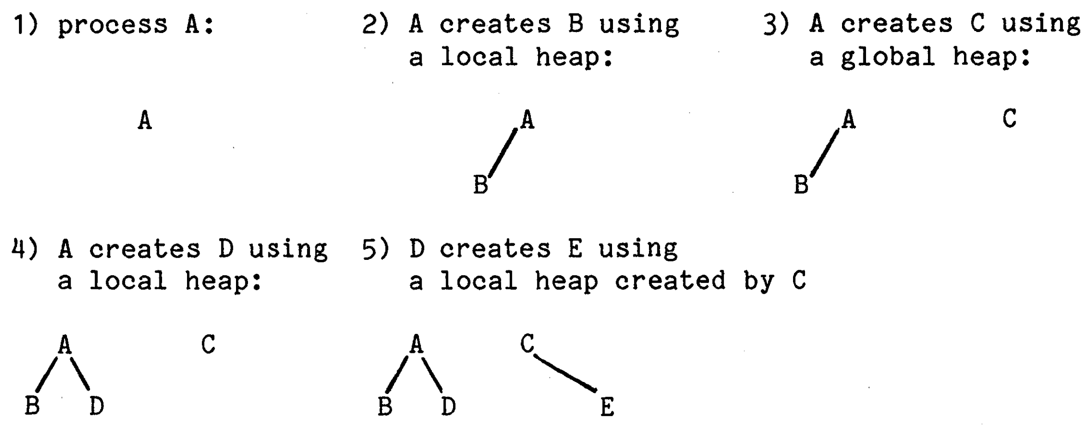

1) process A:
A
2) A creates B using a local heap:
A
B
3) A creates C using a global heap:
A
C
B
4) A creates D using a local heap:
A
C
B
D
5) D creates E using a local heap created by C
A
C
B
D
E

Figure BPM-2. Process Tree Creation

Newly created processes, whether child processes or "roots" (without a parent process) execute asynchronously and possibly simultaneously with the process that creates them. All the processes subordinate to a particular process in a tree are its "descendants" (its children, their children, etc.). Certain restrictions arise naturally out of the process tree structure:

- A process can only have a single parent process.
- A process cannot directly or indirectly be its own descendant.

## GUARDIAN PORTS

Because a process executes asynchronously to its creator process, and may even outlive its creator, it is not able to simply return an error code or raise an exception "to its caller" if it needs attention -- a process has no caller. Instead, process creation specifies an access for a port to which the new process will be sent when it cannot continue execution. Ports and operations on them (sending and receiving messages) are described in Chapter COM, Interprocess Communication. The port access is of type process_port, and only processes may be sent to the port or received from it. The process_port is independent of the tree structure that includes the process and is called the process's guardian. The guardian access cannot be null and must have send rights.

A process is sent to its guardian when the process terminates, is stopped, is destroyed, encounters an error it cannot handle internally, or otherwise needs service. The user of BPM must write software to receive processes from their guardians, evaluate their condition, and take appropriate action. An example of such a "guardian handler" appears at the end of this chapter in the package Guardian Manager Example. A process that receives other processes from a guardian and services them is called the "owner" of the processes routed to the guardian. Figure BPM-3, Guardian Example, illustrates a possible choice of guardians and illustrates the notation used to represent guardian/process relationships.

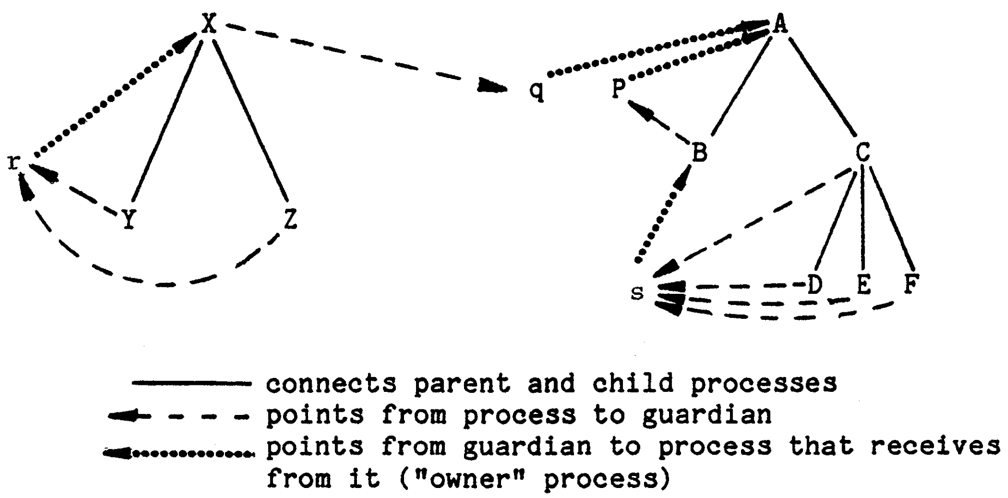

X
q
A
P
r
Y
Z
B
C
s
D
E
F
connects parent and child processes
points from process to guardian
points from guardian to process that receives from it ("owner" process)

Figure BPM-3. Guardian Example

Processes X, Y, and Z represent one process tree, while A through F represent another. Nevertheless, when X terminates or has problems, it goes to guardian q to be handled by process A. The intent might be that process A is responsible for tree X but that operations affecting A should not affect X. For example, tree X might be doing data base operations for process A. If a user stops A, it is important that the data base not be left locked. Thus X is not a part of tree A, but when it completes its work, it "reports back" through its guardian to A. The more typical reporting relationship is indicated by Y and Z reporting through guardian r to their parent X. A minor variation is illustrated by C and its descendants reporting through guardian s to C's sibling B. More complex or more simple relationships can also be constructed.

Some operations on process trees (e.g., Start_tree) can cause multiple processes to be sent to their guardians because of a single operation. When this occurs, the order in which the processes appear at their guardians is undefined, and any program that relies on a particular order is erroneous.

## PROCESS ATTRIBUTES

The attributes of a process are organized into two records, process info and process sched.

Operations are provided to read and modify these records in controlled ways (described in the section Process Information Operations).

The process_info record contains these fields:

name is of type print_name and can be filled in with a character string that identifies the process in messages. This field is not a sure way of distinguishing processes in programs, since two processes can have the same name.

id is of type any_access and may be used for any purpose. The user can use id to link a process to other information about itself.

trace is of type trace type, with these possible values and interpretations:

| | |
|---|---|
| no trace: | normal execution; |
| fault trace: | debug event on every context fault; |
| flow trace: | debug event on control flow changes; |
| full trace: | single-step execution; |

The trace attribute is normally set to values other than no_trace only by the DEBUG-432 debugger.

guardian is of type process_port and is an access with send rights for the port to which the process is sent when a system-recognized condition removes it from the dispatching mix or keeps it from entering the mix. The dispatching mix is the set of processes that are able to run (though they may be idling or waiting at a port).

global as is of type any_access and is an access for the process globals access segment (PGAS) for the process. The PGAS is described in Chapter PEN, Program Environment Access.

state is of type process_state and is described in the section Process States.

The user can only read the global__as or state fields of the process_info record.

## PROCESS STATES

The process state has six fields, but only two conceptually separate parts: condition information and the user stopped count. The five condition information fields change without affecting the user_stopped count. In contrast, changes in the user_stopped_count may change the condition fields, but only if the process was in the ready state.

The set of processes eligible to run is called the dispatching mix, or just the mix. To be in the mix, a process must have condition ready, user_stopped_count <=0, and be explicitly entered into the mix with a Restart operation.

This section first describes the six fields of the process state, and then introduces process state names that are used in the rest of this chapter.

condition is the first field of the process state, and can take on these values:

| | |
|---|---|
| ready | The process is able to run and is either in the mix, able to enter the mix with a Restart operation (user stopped_count <=0), or kept from entering the mix because user_stopped_count >0. |
| user start | The process was in condition ready and a Start operation caused the process's user_stopped_count to change from 1 to 0. |
| user stop | The process was in condition ready and a Stop operation caused the process's user_stopped_count to change from 0 to 1. |
| user restart | The process was in condition ready with user_stopped count >0 and a Restart was issued for the process. |
| service needed | The process has encountered a normal condition requiring attention from its owner. |
| terminated | The process has returned from its initial procedure and has therefore completed operation. |
| error | The process has encountered a potentially recoverable error due to its own operation. |
| fatal error | The process has encountered a fatal error due to its own operation. |
| system error | The process has been irreparably damaged because of a system failure. |
| destroyed | The process has been destroyed (but accesses for it may still exist). |

user stopped count is the second field, an integer which, if greater than zero gives the number of Start operations required on the process before it can enter the mix, and if less than or equal to zero is one minus the number of Stop operations required on the process to remove it from the mix (or keep it from entering the mix).

out of mix is the third field, a boolean that indicates whether the process is excluded from the dispatching mix. When out_of_mix is false, the process is in the mix. To enter the mix, and have out_of_mix set to false, a process must be in condition ready, have user_stopped_count less than or equal to zero, and have the field destroy_in_progress be false.

error info is the fourth field, which gives a specific reason for a process to be in condition error, system_error, or fatal_error, and can have these values:

| | |
|---|---|
| no error | (condition is not error, system_error, or fatal error) The process has not encountered an error. |
| abandoned | (condition = error) All accesses for the process have been discarded before the process is destroyed. |
| internal confusion | (condition = system_error) The process has entered some meaningless state due to an iMAX error. |
| cxt fault handler | (condition = error) A context fault occurred in the context fault handler. |
| no exception handler | (condition = fatal_error) This value is intended to designate the situation where an Ada exception is raised in a process and the process has no handler for the exception raised. Ada exceptions are not yet implemented by the 432 Ada compiler. While exceptions and handling of them will compile, Raise statements generate no code. However, within iMAX, all Raise statements have been replaced by calls to an internal iMAX procedure that sends the calling process to its guardian in state (fatal_error, *, true, no_exception_handler, *). |
| object damaged | (condition = system_error) The process has been irreparably damaged due to a memory or secondary storage failure. |

service info is the fifth field, which gives a specific reason for a process to be in condition service_needed, and can have these values:

| | |
|---|---|
| no service needed | (condition is not service_needed) The process does not need service. |
| schedule | The process has had its periods value go to 0 and can no longer be scheduled to run on a processor. |

| | |
|---|---|
| memory | The process has attempted to allocate memory beyond that which is currently available in the global heap SRO that it is using. Note: iMAX V2 does not use this value in the process state. In V2, a process that requests too much memory waits indefinitely for it (and its process state will still be ready and in the mix). |

destroy in progress is the sixth and last field, a boolean that is true if a destroy operation has been started for the process.

The process state is defined to be a 6-tuple: (condition, user_stopped count, out_of_mix, error_info, service_info, destroy_in_progress). In the descriptions that follow, the error_info, service_info, and destroy in_progress fields are omitted (and a 3-tuple is given) unless they are specifically relevant to the discussion. The symbol "*" is used in any component of the state to designate that any value is permissible. The shorthand "n > 0" or "n <= 0" is used to indicate that any user_stopped count value greater than zero, or less than or equal to zero, respectively, is allowed. Process state transitions are connected with the symbol "=>", for example:

(ready, *, *) => (error, *, true, abandoned, no_service_needed, false)

is the transition when a ready process is found by the iMAX garbage collector to have no references to it.

Table BPM-2, below, lists the names used in this manual for process states. The differences between the process state names used here and the process condition field are that three READY states are distinguished (READY_STOPPED, READY_TO_RESTART, READY_IN_MIX) and that states with destroy_in_progress have the same name as states with condition destroyed.

Table BPM-2. Process State Names

| State | Name |
|---|---|
| (ready, *, *) | READY |
| (ready, n > 0, true) | READY_STOPPED |
| (ready, n <= 0, true) | READY_TO_RESTART |
| (ready, n <= 0, false) | READY_IN_MIX |
| (user_start, 0, true) | USER_START |
| (user_stop, 1, true) | USER_STOP |
| (user_restart, n <= 0, true) | USER_RESTART |
| (error, *, true) | ERROR |
| (service_needed, *, true) | SERVICE_NEEDED |
| (terminated, *, true) | TERMINATED |
| (fatal_error, *, true) | FATAL_ERROR |
| (system_error, *, true) | SYSTEM_ERROR |
| (destroyed, *, true) <u>or</u> (*, *, *, *, *, true) | DESTROYED |

## PROCESS STATE TRANSITIONS

When a process is ready and its condition changes, the new condition only indicates one reason for its not being ready. In fact, other conditions may simultaneously apply, or may arise before the first condition is dealt with. In general, once the process condition is not ready, and as long as it stays other than ready, additional conditions are not indicated; they do not cause the process to be sent to its guardian and do not change the condition information in the process state. When the process next enters the mix, any conditions still outstanding will cause it to be again sent to its guardian marked with one of the outstanding conditions. An exception to the rule of ignoring subsequent conditions occurs when a process is destroyed. Destroying a process immediately sets destroy_in_progress to true and eventually causes the process to be sent to its guardian with condition destroyed, losing any previous condition.

Figure BPM-4 illustrates overall process state transitions. Processes are created in the ready state. Either a user operation (Start, Stop, Restart) or a system-detected condition (e.g., process requests too much memory) can place the process in a non-ready, but recoverable, state (ERROR, SERVICE_NEEDED, USER_START, USER_STOP, USER_RESTART). The process's owner must receive the process from the guardian, and can choose to return such a process to a ready state by calling Read_info and_reset_condition. At least a Restart operation is still required on the process to enter it in the mix. In contrast, when a process is in the states TERMINATED, SYSTEM ERROR, or FATAL ERROR, there is no way to make the process ready again -- it can only be examined and destroyed. Read_info_and_reset_condition can be called for such processes, but will not reset their condition to ready.

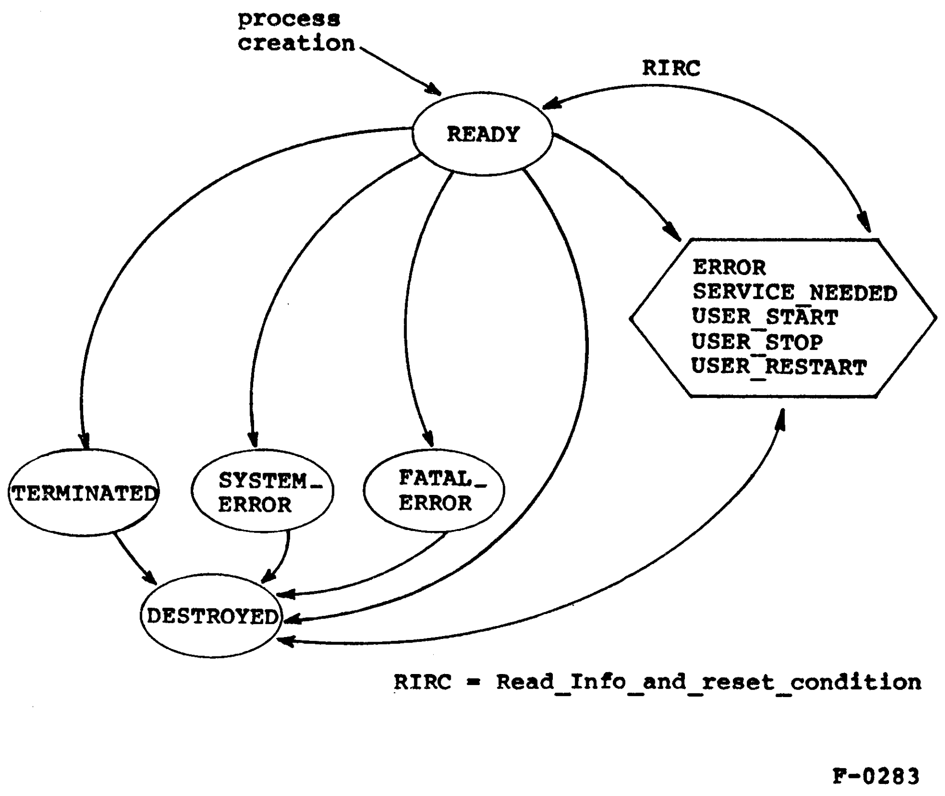


Figure BPM-4. Overall Process State Transitions

Once a process is destroyed, there is no way to revive it; once all accesses for the destroyed process are reclaimed or deleted, it is reclaimed by iMAX garbage collection.

Figure BPM-5 illustrates the detailed process state transitions resulting from the interaction of process state and user Start, Stop, and Restart operations. Processes are created in the READY_STOPPED state, with user stopped_count = 1. Three operations are required to enter the process into the mix. A Start operation that decrements the user_stopped_count from 1 to 0 sends the process to its guardian in state USER_START. The process must be received from its guardian and the Read_info_and_reset condition procedure invoked for it. This operation makes the process ready, now in the READY_TO_RESTART state. A Restart operation then enters the process into the mix. Note that Start operations on processes with user_stopped_count > 1 and Stop operations on processes with user_stopped_ count < 0 do not cause a state transition (though they do change the user_stopped_count field).

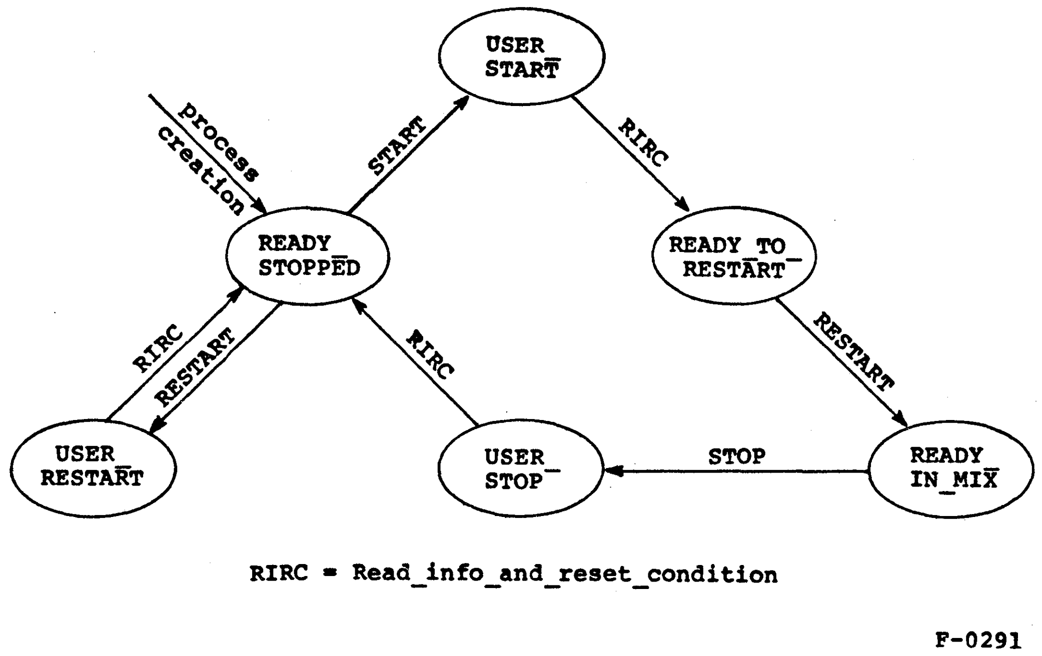


Figure BPM-5. Detailed Process State Transitions

## PROCESS SCHEDULING

The 432 architecture divides process scheduling responsibility between hardware and software. Hardware provides the mechanism for automatic and efficient short-term scheduling using 432 dispatching ports. Software sets the parameters for short-term scheduling and can modify these parameters to implement long-term scheduling policies.

Processes are dispatched to execute on processors in deadline-within-priority order. Higher priority processes dispatch before lower priority processes. Within a given priority, the process with the smallest deadline value dispatches first. Deadline values are computed as the sum of the deadline field in the process_sched record and the time the process arrives at the dispatching port. Thus, if process A and B have the same priority, A's deadline field contains a 5 and B's deadline field contains a 10, but B arrives at the dispatching port 6 time units before A, then B is dispatched before A.

Process scheduling uses the notion of a system time unit, which is a system-wide time unit for process scheduling and timing. The duration of a system time unit is a hardware configuration parameter. For 432/600 systems, a system time unit is 256 microseconds.

Process scheduling parameters are fields in the process sched record:

service is the maximum number of system time units in a service period for the process, i.e., the maximum time that the process may execute before being suspended to be redispatched, possibly allowing other processes to run. Service is the process's "time slice". When a process is suspended and redispatched because its service period expires, it is still in state (ready, n $\leq$ 0, false). There is no way to specify an unlimited service period. If service is zero, then the actual service period is $2^{16}$ (65,536) system time units. Service is of type short_ordinal.

periods is the number of service periods remaining for this process. Periods is set by user software and is the number of times remaining that the process will be dispatched without service by the process's owner. However, if periods is set to the value of the constant infinite_period_count (65,535), then it is never decremented, and the process is never sent to its guardian for rescheduling. Otherwise, periods is decremented each time the process is dispatched, whether the redispatching is because of an expired service period or because the process has been blocked at a communication port or has idled. When periods is decremented from 1 to 0, the process is sent to its guardian with state (service_needed, n $\leq$ 0, true, no_error, schedule, false), instead of being dispatched. To return the process to the mix, its owner must invoke Set_sched_params to adjust the process_sched record, and then call Restart on the process. Periods is of type short_ordinal.

time is the total accumulated process execution time in system time units. Time is incremented only if the process is actually running on a processor when the system time unit "ticks" (when the PCLK signal to the processor is asserted). Time is of type ordinal.

deadline is an indication of how long a process should have to wait for dispatching relative to other processes of the same priority. If two processes with the same priority arrive at a dispatching port at the same time, the one with the smaller value of deadline is dispatched first. Deadline is of type deadline_type, a subrange of short_ordinal.

priority is the current dispatching priority of the process. When processes with different priorities are at a dispatching port, a process with the highest priority is always dispatched first. Priority is of type short_ordinal.

## PROCESS TYPE RIGHTS

Process accesses have an iMAX-defined type right, control rights. This right permits the process to be passed to the process control operations and to have its parameters set by the process information operations. It also affects the information returned when process information is read (described under Process Information Operations below). Whenever a process is received from its guardian, the received access has control rights.

## PROCESS OPERATIONS

iMAX provides operations to create and destroy processes, to move processes in and out of the dispatching mix, and to read and modify process attributes in controlled ways.

The attributes of a process are passed to the process operations in the process_info and process_sched records described above. Some operations use only certain of the fields in these records. The operation descriptions below specify which fields are actually used.

### PROCESS CREATION

The Create function creates a process and returns an access for the new process, given:

- initial procedure and its parameter
- process info and scheduling attributes
- an optional heap SRO access
- optional sizes for things associated with the process

If no heap SRO access is specified, the default_global_heap_SRO specified by the caller's process globals access segment (PGAS) is used to create the process. The PGAS is described in Chapter PEN, Program Environment Access. If a local heap SRO access is specified, the process which created that local heap SRO must not be in terminated, system_error, fatal_error, or destroyed condition, or the condition error exception is raised.

The initial procedure is an instance of the package type procedure val, and takes one in parameter, init params, of type any access. As part of creating a process, iMAX creates a context object for the activation of the initial procedure. When the initial procedure returns, control returns to iMAX, which sends the process to its guardian in terminated condition.

The process_info and process_sched records communicate needed process attributes. Only the name, id, trace, and guardian fields are used in the process_info record. The guardian port access must have send rights, or the no_send_rights_on_guardian exception is raised. Only the service, periods, deadline, and priority fields are used in the process_sched record.

The heap SRO access used must have create rights, or the Ada constraint error exception is raised. If a local heap SRO is specified, it indicates that the new process is a child of the process that created the local heap SRO. If a global heap SRO is specified, the new process is made the root of a new process tree. Objects associated with the new process are allocated both from the heap SRO specified in calling Create and from the global heap SRO with the same memory type. Processes can be created to run in either frozen or normal memory, depending on the memory type of the SRO specified in calling Create. SROs and memory types are described in Chapter STO, Storage Management.

The user may optionally specify: the number of access descriptors in the process globals access segment, the number of entries in the initial process stack object table, and the number of bytes in the initial process stack allocation block (which is not limited by the maximum size segment). If any of these three values is omitted or is less than an iMAX-specified minimum default value, then the iMAX-supplied value is used and the user-specified value is ignored. Note that if a process exhausts either its initial object table or its initial stack allocation block that iMAX storage management will automatically allocate additional object table or stack space for the process.

When the process is created, a process globals access segment is created and its system-recognized fields are filled in. An access for the new process with control rights is returned.

## PROCESS CONTROL OPERATIONS

Process control operations are used to make processes ready to enter the dispatching mix, to start and stop processes by changing the user stopped__count, and to enter eligible processes into the mix by assigning out_of_mix false.

All process control operations take a process access parameter which must have control rights (or the Ada constraint_error exception is raised). Some process control operations place restrictions on the state of the process that is passed to them. If these restrictions are violated, the condition_error exception is raised.

The Read info and reset condition operation returns the process_info record for the process and, if the process state is not DESTROYED, TERMINATED, FATAL_ERROR, or SYSTEM_ERROR, resets the condition field to ready, error_info to no_error, and service_info to no_service_needed. This procedure is designed to be called for a process immediately after an access for the process is received from its guardian by the process's owner. Note that this procedure is the only way that an existing process can be placed in the READY state. Read_info_and_reset condition can raise the conflicting user access exception, described in the section Process Information Operations below. If this exception occurs, the operation can be retried later. The passed process should not be overflow enqueued on its guardian port when this procedure is called, or the guardian port overflow exception may be raised. This exception can be avoided by calling Read_info_and_reset_condition only on processes that have just been received from their guardians.

The Start tree procedure takes a process which must be in a state other than DESTROYED, TERMINATED, FATAL_ERROR, or SYSTEM_ERROR. The passed process is started, and also each of its descendant processes which is in some state other than DESTROYED, TERMINATED, FATAL_ERROR, or SYSTEM ERROR. The user_stopped_counts of the affected processes are decremented. If, as a result, any of them reach zero and the process is ready, then the corresponding process is sent to its guardian in condition user_start to inform its owner that it needs to be Restarted. The Start descendants procedure takes a process which must be in some state other than DESTROYED and starts each of its descendant processes which is in some state other than DESTROYED, TERMINATED, FATAL_ERROR, or SYSTEM_ERROR.

The Stop tree procedure takes a process which must be in some state other than DESTROYED. The passed process is stopped, and also each of its descendant processes which are in some state other than DESTROYED. The user_stopped_counts of the affected processes are incremented. If, as a result, any of the user_stopped_counts reachs one and the process is in the mix, then the process is sent to its guardian in condition user_stop. The Stop descendants procedure takes a process which must be in some state other than DESTROYED and stops each of its descendant processes which are in some state other than DESTROYED. The effect of stopping processes may be asynchronous to the caller. That is, "stopped" processes may not actually leave the dispatching mix and appear at their respective guardians until after the caller has been returned to.

The process state transitions when a process is stopped is:

```
    (ready, 0, *)  =>  (user_stop, 1, true)
    (x, n, *)      =>  (x, n+1, *)
```

The Restart procedure enters a single ready process with user_stopped count <= 0 which is out of the dispatching mix into the dispatching mix. If user_stopped_count > 0, the process is sent to its guardian with condition = user_restart to inform the process owner that one or more Starts are needed for the process before it can enter the mix. The process state transitions are:

```
    (ready, n, true)    =>   (user_restart, n, true) for n <= 0
    (ready, n, true)    =>   (ready, n, false)        for n > 0
```

Any other state in the passed process raises the condition_error exception in the caller.

## PROCESS DESTRUCTION

The Destroy_tree and Destroy_descendants procedures destroy processes. Both take a single parameter, a process access. The process access must have control rights, or the Ada constraint_error exception is raised. Destroy_tree destroys the passed process and all of its descendants, while Destroy_descendants destroys only the descendants and not the passed process. Note that it is impossible to destroy a process without destroying all of its descendants.

Process destruction may be asynchronous to the caller -- the affected processes may not actually have their condition set to destroyed until some time after control returns to the caller. However, the destroy_in progress flag in the process state is set for all affected processes before control returns to the caller, and stays true even after the condition is set to destroyed. When processes do become destroyed, they are sent to their respective guardians with the overall process state transition:

```
    (*, *, *, *, *, *)  =>  (destroyed, *, true, no_error,
                             no_service_needed, true)
```

Once a process has been destroyed, it cannot be revived or restarted. A destroyed process is reclaimed by the system whenever it becomes garbage.

The only operations allowed on a process with condition = destroyed or destroy_in_progress are Read_info_and_reset_condition, Read_info, and Read_sched_params.

## PROCESS INFORMATION OPERATIONS

BPM synchronizes access to the process_sched and process_info attributes. If a user process is delayed for some reason while performing an operation on the process attributes, another process requesting a process information operation may raise a conflicting_user_access exception. The request may be tried again later.

The Read_info_and_reset_condition (RIRC) procedure is both a process information operation and a process control operation. The RIRC procedure is described above, in the section Process Control Operations.

The procedures Set_sched_params and Set_info selectively set some or all of the respective process attributes for the designated process. Each requires control rights on the passed process. Each takes as parameters the process to be affected, a record containing the new attribute values, and a record of boolean values indicating which particular scheduling or information attributes are to be set. The new values of the attributes will not take effect until the process next reaches a convenient point in the dispatching cycle. The new values will, however, be returned by any subsequent call to read them. Set sched_params returns the previous value of periods for accounting purposes. The process state and global_as may not be set by Set_info.

The functions Read sched params and Read info read the respective process attributes. Each takes as a parameter the process of interest. No rights are necessary on calls to Read_sched_params. On calls to Read_info, if control rights are present on the passed process, then all the attributes are returned. If the passed process does not have control rights, then nulls are returned for the id, trace, guardian, and global_as attributes and only the name and state values are returned.

The Get parent function returns the process above the passed process in the tree. If the passed process has no parent (is the root of a tree) or is destroyed, then a null access is returned. The returned access has control rights only if it is not null and if the passed parameter has control rights.

The Get descendants function returns a list of the direct descendants (child processes) of the passed process. The passed process must not be destroyed (or the condition_error exception is raised). The returned access is of type process list, an access for an array of process accesses. The returned access is null if the process has no descendants, else an access is returned for the array. The number of array elements equals the number of child processes. The accesses in the array have control rights only if the passed parameter has control rights. If the designated process has too many descendants to be handled by Get_descendants, then the too many descendants exception is raised.

## EXCEPTIONS

The exceptions that can be raised by each process operation are tabulated in Table BPM-3.

Table BPM-4. Process Management Operations and Exceptions

| OPERATION: | EXCEPTION: CONSTRAINT ERROR | NO_SEND_RIGHTS_ON_GUARDIAN | CONDITION_ERROR | CONFLICTING_USER_ACCESS | GUARDIAN_PORT_OVERFLOW | TOO_MANY_DESCENDANTS |
|---|---|---|---|---|---|---|
| CREATE | | X | X | | | |
| READ_INFO_AND_RESET_CONDITION | X | | | X | X | |
| START_TREE | X | | X | | | |
| START_DESCENDANTS | X | | | | | |
| STOP_TREE | X | | X | | | |
| STOP_DESCENDANTS | X | | | | | |
| RESTART | X | | X | | | |
| DESTROY_TREE | X | | | | | |
| DESTROY_DESCENDANTS | X | | | | | |
| READ_INFO | | | | X | | |
| SET_INFO | X | | | X | | |
| READ_SCHED_PARAMS | | | X | X | | |
| SET_SCHED_PARAMS | X | | X | X | | |
| GET_PARENT | | | | | | |
| GET_DESCENDANTS | | | X | | | X |

F-0376

## BASIC PROCESS MANAGEMENT PACKAGE

```
with Descriptor_Definitions, Typed_Ports, iMAX_Definitions,
     Process_Globals_Definitions;
package Basic_Process_Management is

  -- Function:
  --   Basic Process Management provides functions to create, start,
  --   stop, alter, and destroy processes.  It provides a default tree
  --   structure for all processes which is used to direct these
  --   functions.  It associates with each process a guardian port
  --   where the process is sent when it exits or cannot enter the
  --   dispatching mix.


  use iMax_Definitions;



  -- EXCEPTIONS

  condition_error:            exception;
      -- A process is not in the condition or state required by
      -- the procedure to which it was passed.

  too_many_descendants:       exception;
      -- The passed process has too many descendants to be returned
      -- by the get descendants procedure.

  guardian_port_overflow:     exception;
      -- Read_info_and_reset_condition was called on a process which
      -- is overflow enqueued on the guardian port.

  conflicting_user_access:    exception;
      -- The user is requesting Set_sched_params, Read_sched_params,
      -- Read_info, or Read_info_and_reset_condition in a conflicting
      -- manner.

  no_send_rights_on_guardian: exception;
      -- The guardian port passed in the info parameter to Create does
      -- not have send rights.
-- TYPES

type out_of_mix_condition is (
  ready,                -- capable of running
  terminated,           -- returned from initial procedure
  destroyed,            -- destroyed by owner but object references for
                        -- it may remain
  system_error,         -- irreparably damaged due to a system failure
  fatal_error,          -- unrecoverable error due to process' own
                        -- operation
  error,                -- potentially recoverable error due to process'
                        -- own operation
  service_needed,       -- normal condition requiring attention from
                        -- owner, such as scheduling
  user_stop,            -- a stop operation caused the process'
                        -- user_stopped_count to change from 0 to 1
  user_start,           -- a start operation caused the process'
                        -- user_stopped_count to change from 1 to 0
  user_restart);        -- a restart was issued for the process when
                        -- its user_stopped_count was > 0

type error_type is (
    -- Additional information applicable to error, fatal_error, and
    -- system_error.
  no_error,
  abandoned,                 -- There are no reference paths to the
                             -- process but it is not destroyed (error).
  internal_confusion,        -- A system error has rendered the process
                             -- unusable (system_error).
  cxt_fault_handler,         -- A context-level fault occurred in the
                             -- context-level fault handler (error).
  no_exception_handler,      -- There is no exception handler in the
                             -- process (fatal_error).
  object_damaged);           -- The process is irreparably damaged due to
                             -- a memory or secondary storage failure
                             -- (system_error).

type service_type is (
    -- Additional information applicable to service_needed.
  no_service_needed,
  schedule,   -- The process has run for its alloted number of service
              -- periods and needs rescheduling.
  memory);    -- The process has requested more memory than is
              -- available to it.
  type process_state is
    record
      condition:          out_of_mix_condition;
      user_stopped_count: integer;
      out_of_mix:         boolean;
      error_info:         error_type;
      service_info:       service_type;
      destroy_in_progress: boolean;
    end record;

  initial_process_state: constant process_state :=
    process_state'(ready, 1, true, no_error, no_service_needed, false);

  subtype deadline_type is iMAX_Definitions.deadline_scheduling_value
                             range 0 .. 2**14 - 1;

  type process_sched is
    record
      service:  short_ordinal;  -- service period
      periods:  short_ordinal;  -- period count
      time:     ordinal;        -- process clock
      deadline: deadline_type;  -- deadline and priority are parameters
      priority: short_ordinal;  -- to deadline-within-priority
                                -- scheduling
    end record;

  infinite_period_count: constant short_ordinal := 16#FFFF#;
      -- When a process has this as its process_sched.periods value,
      -- the process is never stopped for scheduling.

  package Process_Port_Def is
    new Typed_Ports.Simple_Port_Def(process);

  subtype process_port is Process_Port_Def.user_port;

  null_process_port: process_port renames
                       Process_Port_Def.null_user_port;

  type trace_type is (
    no_trace,
    fault_trace,
    flow_trace,
    full_trace);
```

```
type process_info is
  record
    name:      print_name;
    id:        any_access;
                 -- for the user to use
    trace:     trace_type;
    guardian:  process_port;
                 -- The process is sent here when it exits or cannot
                 -- enter the dispatching mix.
    global_as: Process_Globals_Definitions.process_globals_rep;
                 -- This will likely be retyped from an access segment
                 -- which contains a process_globals_rec followed
                 -- by user-defined access descriptors.
    state:     process_state;
  end record;

package type procedure_val is
  -- The initial procedure supplied by the user for each process
  -- is an instance of this package type.
  procedure proc(
      params: any_access);
end procedure_val;


max_process_list_length: constant := 64;

type process_list_rep is
  record
    num_processes: short_ordinal;
    processes: array(short_ordinal range 1 ..
                 max_process_list_length) of process;
  end record;

type process_list is access process_list_rep;
type which_sched is
  record
    set_service:  boolean;  -- true if service is to be set
    set_periods:  boolean;  -- true if periods is to be set
    set_deadline: boolean;  -- true if deadline is to be set
    set_priority: boolean;  -- true if priority is to be set
  end record;

type which_info is
  record
    set_name:     boolean;  -- true if name is to be set
    set_id:       boolean;  -- true if id is to be set
    set_trace:    boolean;  -- true if trace is to be set
    set_guardian: boolean;  -- true if guardian is to be set
  end record;

subtype table_size_type is short_ordinal range 0 .. 2**12 - 1;
  -- This type is used to limit the acceptable values for
  -- the init_stack_objtab_size parameter to Create.

subtype mem_size_type is ordinal range 0 ..
                          Descriptor_Definitions.max_physical_mem_sz;
  -- This type is used to limit the acceptable values for
  -- the init_stack_size parameter to Create.


-- PROCEDURES

function Create_process(
    init_proc:            procedure_val;
                            -- Initial procedure for the created
                            -- process to execute.
    init_params:          any_access;
                            -- Parameters to the initial procedure.
    info:                 process_info;
                            -- Info attributes for created process.
                            -- (Only the name, id, trace, and guardian
                            -- fields are used.)
                            -- The guardian must have send rights.
    sched:                process_sched;
                            -- Scheduling parameters for created
                            -- process (Only the service, periods,
                            -- deadline, and priority fields are
                            -- used.)
    heap_SRO:             storage_resource := null;
                            -- SRO from which new process is to be
                            -- created.  Default is the global heap
                            -- SRO found in the calling process'
                            -- globals access segment.
    global_as_size:       short_ordinal := 0;
                            -- Size (# of access descriptors) of
                            -- process globals access segment.  If
                            -- too small a value is passed, it will
                            -- be replaced by the smallest
                            -- allowable size.
    init_stack_objtab_size: table_size_type := 0;
                            -- Initial size of the process stack
                            -- object table.  If too small a value
                            -- is passed, it will be replaced by
                            -- the smallest allowable size.
    init_stack_size:      mem_size_type := 0)
                            -- Initial size of the process stack
                            -- allocation block.  If too small a
                            -- value is passed, it will be replaced
                            -- by the smallest allowable size.
  return process;  -- The newly created process (has control rights,
                   -- but not read or write rights).

  -- Function:
  --   The process which created the given heap_SRO must not be in
  --   terminated, destroyed, system_error, or fatal_error condition.
  --   Creates and returns a process in state initial_process_state.
  --   Its attributes are set according to init_proc, init_params,
  --   info, and sched.  The new process is made a direct descendant
  --   of the process which created the given heap_SRO, unless that
  --   heap_SRO is a global heap or is not given, and thus defaults
  --   to a global heap.  In these cases, the new process is made the
  --   root of an entirely new process tree.  A process globals
  --   access segment is created for the new process and its
  --   system-recognized fields are filled in.

  --   Exceptions:
  --     no_send_rights_on_guardian
  --     condition_error
```

```
procedure Destroy_tree(
    p: process);  -- Identifies the tree of processes to be destroyed.
                  -- Must have control rights.

  -- Function:
  --   The given process and its descendants are destroyed (after
  --   first being stopped if they are in the dispatching mix).  The
  --   effect of this operation may be asynchronous to the caller.
  --   That is, the affected processes may not actually become
  --   destroyed until some time after the caller has been returned
  --   to.  When they do become destroyed, they are sent to their
  --   guardians.  When the destroyed processes become garbage, they
  --   are reclaimed by the system.

  --   Exceptions:
  --     constraint_error


procedure Destroy_descendants(
    p: process);  -- Process whose descendants are to be destroyed.
                  -- Must have control rights.

  -- Function:
  --   The descendants of the given process are destroyed in the same
  --   manner as for Destroy_tree.
  --   Effects of this operation may be asynchronous to the caller.

  --   Exceptions:
  --     constraint_error


procedure Start_tree(
    p: process);  -- Identifies the tree of processes to be started.
                  -- Must have control rights and must not be in
                  -- terminated, destroyed, system_error, or
                  -- fatal_error condition.

  -- Function:
  --   For the given process and each of
  --   its descendants which are not in terminated, destroyed,
  --   system_error, or fatal_error condition, the user_stopped_counts
  --   are decremented by 1.  If, as a result, any of
  --   them reach 0, the corresponding processes are sent to their
  --   guardians with condition = user_start.
  --   Note that a process in terminated, system_error, fatal_error,
  --   or destroyed condition inhibits the Start operation on all of
  --   its descendants, as well as on itself.

  --   Exceptions:
  --     constraint_error
  --     condition_error
```

```
procedure Start_descendants(
    p: process);  -- Process whose descendants are to be started.
                  -- Must have control rights.  (Its condition is
                  -- irrelevant.)

  -- Function:
  --   The descendants of the given process are started in the same
  --   manner as for Start_tree.

  --   Exceptions:
  --     constraint_error


procedure Restart(
    p: process);  -- Process to be restarted.
                  -- Must have control rights, must be in condition
                  -- ready, and must be out_of_mix.

  -- Function:
  --   If user_stopped_count <= 0, out_of_mix is assigned false and
  --   the process is started.  If user_stopped_count > 0, the
  --   process is sent to its guardian with condition = user_restart.

  --   Exceptions:
  --     constraint_error
  --     condition_error


procedure Stop_tree(
   p: process);  -- Identifies the tree of processes to be stopped.
                 -- Must have control rights and must not be
                 -- destroyed.

  -- Function:
  --   For the given process and its descendants which
  --   are not destroyed, the user_stopped_counts are
  --   incremented by 1.  If, as a result, any of them reach
  --   1, the corresponding process is stopped and sent to its
  --   guardian.  The effect of this operation may be asynchronous to
  --   the caller.  That is, the affected processes may not actually
  --   become stopped until some time after the caller has been
  --   returned to.

  --   Exceptions:
  --     constraint_error
  --     condition_error
```

```
procedure Stop_descendants(
    p: process);  -- Process whose descendants are to be stopped.
                  -- Must have control rights.  (Its condition is
                  -- irrelevant.)

  -- Function:
  --   The descendants of the given process are stopped in the same
  --   manner as for Stop_tree.  The effects of this operation may be
  --   asynchronous to the caller.

  --   Exceptions:
  --     constraint_error


function Set_sched_params(
    p: process;           -- Process whose scheduling parameters are to
                          -- be set.  Must have control rights and must
                          -- not be destroyed.
    w: which_sched;       -- Which scheduling parameters to set.
    s: process_sched)     -- New scheduling parameters.
  return short_ordinal;   -- Previous value of periods.

  -- Function:
  --   Selected scheduling parameters of the given process are reset
  --   to take on the given values.  The parameter w indicates which
  --   parameters are to be set.  The new values do not take effect
  --   until the process reaches a convenient point in the
  --   dispatching cycle.  The new values are, however, returned by
  --   any subsequent call to read them.  The previous value of the
  --   periods field is returned for accounting purposes.

  --   Exceptions:
  --     constraint_error
  --     conflicting_user_access
  --     condition_error


function Read_sched_params(
    p: process)            -- Process whose scheduling parameters are
                           -- to be returned.  Must not be destroyed.
  return process_sched;    -- Scheduling parameters of p.

  -- Function:
  --   The scheduling parameters of the given process are returned.

  --   Exceptions:
  --     conflicting_user_access
  --     condition_error
```

```
procedure Set_info(
    p: process;         -- Process whose info attributes are to be set.
                        -- Must have control rights.
    w: which_info;      -- Which parameters to set.
    i: process_info);   -- New info attributes.

  -- Function:
  --   Some or all of the name, id, trace, and guardian info
  --   attributes of the given process are reset to take on the given
  --   values.  The parameter w indicates which parameters are to be
  --   set.  The new values do not take effect until the process
  --   reaches a convenient point in the dispatching cycle.
  --   They are returned by any subsequent call to read them, however.

  --   Exceptions:
  --     constraint_error
  --     conflicting_user_access


function Read_info(
    p: process)          -- Process whose info attributes are to be
                         -- returned.
  return process_info;   -- Info attributes of the given process.

  -- Function:
  --   The info attributes of the given process are returned.  If the
  --   given process does not have control rights, then null values
  --   are returned for the id, guardian, and global_as attributes.

  --   Exceptions:
  --     conflicting_user_access


function Read_info_and_reset_condition(
    p: process)         -- Process whose state info is to be returned.
                        -- Must have control rights.
  return process_info;  -- Info attributes of the given process.

  -- Function:
  --   The info attributes of the given process are returned.  If
  --   it is not in destroyed, terminated, system_error, or
  --   fatal_error condition, its condition is reset to ready, and
  --   error_info and service_info are set to no_error and
  --   no_service_needed.

  --   Exceptions:
  --     constraint_error
  --     conflicting_user_access
  --     guardian_port_overflow
  function Get_descendants(
      p: process)          -- Process whose descendants are to be
                           -- returned.  Must not be destroyed.
    return process_list;   -- Descendants of p.

    -- Function:
    --   The direct descendants of the given process (i.e., one level in
    --   the process tree) are returned.  The returned processes have
    --   control rights if and only if the given process has control
    --   rights.  The states of the processes are not affected by this
    --   operation.

    --   Exceptions:
    --     condition_error
    --     too_many_descendants


  function Get_parent(
      p: process)     -- Process whose parent is to be returned.
    return process;   -- Parent of p.

    -- Function:
    --   The parent of the given process (i.e., the process above it in
    --   the tree) is returned.  The returned process has control
    --   rights if and only if the given process has control rights.
    --   If the given process is the root of a tree (i.e., has no
    --   parent) or is destroyed, then a null value is returned.
    --   The states of the processes are not affected by this operation.

    --   Exceptions:
    --     none


  -- PROCESS RIGHTS

  control_rights: constant Descriptor_Definitions.rights :=
                            Descriptor_Definitions.type_right_1;
      -- Only one system right, control_rights, is defined for
      -- processes.  It is required on processes passed as parameters
      -- to the process control operations; otherwise, the
      -- constraint_error exception is raised.

end Basic_Process_Management;
```

## GUARDIAN MANAGER EXAMPLE PACKAGE

```
with Basic_Process_Management;
package Guardian_Manager_Example is:

  -- Function:
  --   This package provides a single procedure, Guardian_handler,
  --   that takes a guardian port as its parameter and handles
  --   processes arriving at the guardian.

  --   This is a general-purpose guardian handler usable for many
  --   applications of iMAX Basic_Process_Management.


  procedure Guardian_handler(
    g:  Basic_Process_Management.process_port);

    -- Function:
    --   Processes are received from the guardian, their condition
    --   is reset, and further action is taken based on their
    --   previous condition:

    --     A terminated process is destroyed.

    --     A destroyed process is ignored (it will be reclaimed
    --     by garbage collection when all refs to it are gone).

    --     A process in condition user_stop, user_start, or
    --     user_restart is restarted if its user_stopped_count
    --     <= 0.

    --     A process in any other condition triggers an error
    --     message to the system console device.

    --     This handler does not reschedule processes with period
    --     counts expired.  This is treated as an error condition
    --     and triggers a message to the console.

    -- Warning:
    --   This procedure never returns to its caller.

end Guardian_Manager_Example;
```

## GUARDIAN MANAGER EXAMPLE PACKAGE BODY

```
with Unchecked_conversion, iMAX_Definitions,
     Basic_Process_Management, IO_Devices;
package body Guardian_Manager_Example is


  package BPM renames Basic_Process_Management;

  use iMAX_Definitions;


  -- used for writing error messages
  type buffer_rep is array(1..80) of character;
  type buffer_type is access buffer_rep;



  function Retype_buffer_to_any_ds
    -- (a:  buffer_type)
    -- return any_ds;
    is new Unchecked_conversion(buffer_type, any_ds);

    -- Function:
    --   The given access for a buffer is retyped to any_ds.
    --   This operation does not effect the rights on the access.


  procedure Write_error(
      info:  BPM.process_info)
  is

    -- Function:
    --   The process described by info is in an erroneous state.
    --   Write_error writes a descriptive message to the system
    --   console.  The process should be in condition ready,
    --   system_error, fatal_error, error, or service_needed.

    use BPM;

    buffer:  buffer_type;

  begin
    buffer := new buffer_rep(others => ' ');
    buffer(1..27) := "**PROCESS ERROR: CONDITION=";
    case info.state.condition is
      when ready =>
        buffer(28..34) := "READY**";
      when system_error =>
        buffer(28..41) := "SYSTEM_ERROR**";
      when fatal_error =>
        buffer(28..40) := "FATAL_ERROR**";
      when error =>
        buffer(28..34) := "ERROR**";
      when service_needed =>
        buffer(28..43) := "SERVICE_NEEDED**";
      when others =>
        null;
    end case;

    IO_Devices.console.write(Retype_buffer_to_any_ds(buffer),0,80);

    buffer(1..80) := (others => ' ');
    buffer(1..12) := "**PRCS NAME=";

    for i in 1..15 loop
      buffer(i+12) := info.name(i);
    end loop;

    IO_Devices.console.write(Retype_buffer_to_any_ds(buffer),0,80);

  end Write_error;


  procedure Guardian_handler(
      g:  BPM.process_port)
    is

    use BPM;

    p:     process;          -- current process
    info:  BPM.process_info; -- info for current process
  begin
    loop
      BPM.Process_Port_Def.Receive(g, p);
      info := BPM.Read_info_and_reset_condition(p);
      case info.state.condition is
        when terminated =>
          BPM.Destroy_tree(p);
        when destroyed =>
          null;
        when user_stop | user_start | user_restart =>
          if info.state.user_stopped_count <= 0 then
            Restart(p);
          end if;
        when others =>
          Write_error(info);
      end case;
    end loop;
  end Guardian_handler;

end Guardian_Manager_Example;
```

# CHAPTER COM
# INTERPROCESS COMMUNICATION

This chapter describes the iMAX packages which manage and enhance the 432 architecture's interprocess communication (port) mechanism. These packages manage the following object types:

| | |
|---|---|
| ● messages | any 432 object for which an access descriptor is sent from a sending process to a receiving process |
| ● carriers | system objects that carry messages on behalf of processes |
| ● ports | system objects that queue messages and carriers |

The basic operations on these objects are:

- sending a message in a carrier to a port
- receiving a message in a carrier from a port

Both of these operations can invoke a third operation, the forwarding of the carrier to a second port.

iMAX supports a typed view of ports as well as an untyped view. The typed view allows the user to create ports for specified message types; only messages of the specified type can be sent to or received from such a port. Type-checking is provided by the Ada compiler system and incurs no run-time overhead. Both views support both the simple and the enhanced port operations.

In the simple model, processes are the active agents which send and receive messages at ports. Processes wait at ports to send messages if the port is full and to receive messages if the port is empty. In this model, the First-In, First-Out (FIFO) queue of blocked processes waiting at a full port is an unbounded extension of the port's limited message queue. The simple model also supports variants of send and receive which transfer a message only if the operation does not block. A boolean parameter is assigned true or false to indicate success or failure.

In the enhanced model, surrogates can be created which wait to send or receive messages in place of processes. Also, priorities can be associated with surrogates and used for prioritized message enqueuing within ports (but the queue of waiting processes/surrogates is still FIFO). Finally, the enhanced model supports the forwarding of surrogates to a second port after they have completed their first port operation. This capability has some important applications (described in the section Surrogate Applications).

The simple and enhanced models are unified by the concept of carriers for messages. A carrier is associated with each process and represents the process when the process must wait at a port. Users can also create carriers explicitly to act as process surrogates. This chapter reflects the underlying unity of the mechanism by explaining the simple and enhanced operations together.

## MESSAGES

Messages are transferred by copying access descriptors. Figure COM-1 illustrates the steps in transferring a message AD between processes. After a message is sent, both the sender and the receiver have accesses for the message. Delete rights are set on received accesses, just as if they were explicitly copied.

## PORTS

Ports are queuing mechanisms supported by the hardware and consisting of two queues, a bounded message queue and an unbounded carrier queue. The message queue contains the message accesses that have been sent to the port but not yet received. The message queue also contains a queuing value for each message entry that determines where it is inserted in the queue. The queuing value is either 0 (for simple port operations or FIFO ports) or is obtained from the surrogate carrier used in sending the message. The port's queuing discipline specifies that messages are enqueued either FIFO or by priority. For priority ports, messages with higher priority are enqueued first and queuing is FIFO within the same priority.

The message queue has a maximum number of entries that is fixed when the port is created and cannot be changed. When the number of messages in the queue equals the maximum, the port is said to be full -- when a message is sent to a full port, the operation blocks and the sending carrier must wait in the carrier queue. When the message queue has no entries, the port is said to be empty -- when a receive is executed on an empty port, the operation blocks, and the receiving carrier must wait in the carrier queue.

The carrier queue is an unbounded FIFO queue with two uses. It can contain carriers waiting to receive a message from the port (if the port is empty) or carriers waiting to send a message to the port (if the port is full). Because these cases are mutually exclusive, only one carrier queue is needed.

Type rights for ports are send_rights and receive_rights. Send_rights are required to send a message to the port or to forward a carrier to the port. Receive_rights are required to receive a message from the port.

When a port is created, using Create port, the queuing discipline and number of message entries are specified. The number of message entries must be in the range 1.. max_message_count. The port's message queue and carrier queue are both initially empty. An access is returned with send and receive rights but no representation rights. The user cannot retrieve or modify the attributes of an existing port (e.g., get or change the queuing discipline of an arbitrary port).

## CARRIERS

Carriers transport messages to and from ports. Additionally, surrogate carriers provide message queuing values. A process carrier is implicitly associated with each process but not visible to users. Process carriers are used by the simple Send and Receive operations. If the process carrier blocks at a port, then the associated process blocks. The forwarding of a process carrier sends it to a dispatching port so that its associated process can run.

Users can create surrogate carriers, which act as surrogates on behalf of processes. When a surrogate carrier is created using Create carrier, a number of access slots (in the range 1..max_carrier_size) and a short ordinal priority are specified. The priority is the queuing value used if the surrogate carrier carries a message being sent to a prioritized port. A carrier's priority can also be changed by calling Set carrier priority. Typed carriers also contain an identification value of a user-specified type, specified when the carrier is created. The Get_carrier_id operation returns the identification of a typed carrier.

Type rights for surrogate carriers are use_rights, required to use the carrier in any surrogate operation. When a carrier is created, an access is returned for a refinement of the carrier with all rights. This refinement provides access to the access slots in a carrier. The first access in the refinement references the carried message. The second access (if any) is used by the Typed_Ports package to associate identification information with the carrier. Other slots are user-defined. The user must retype the carrier access to iMAX_Definitions.any_access_array to read or write the visible access slots in the carrier.

## MESSAGE SENDING

Sending a message requires a carrier and two ports. The carrier transports the message to the first port and waits if the port is full. When the message is delivered to the port, the carrier is forwarded to the second port.

Two operations always send messages. Send implicitly uses the sending process's carrier with the dispatching port as the second port. Surrogate send uses an explicitly specified surrogate carrier and second port. For both operations:

1. If the port is full, the message is copied into the sending carrier, which is appended to the FIFO queue of waiting carriers.

2. Otherwise, if carriers are waiting (which now implies that the port is empty), the message is copied into the first waiting carrier, which is removed from the list and forwarded to its second port.

3. Otherwise, the message and queuing value are inserted in the port message queue.

A Send operation in which the process carrier must wait is called a blocking send, and the sending process blocks with its carrier. When space in the message queue eventually becomes available, the blocked process's message is enqueued and the process carrier is dequeued and forwarded to its dispatching port.

A non-blocking Send does not involve the process carrier. By contrast, a surrogate carrier is forwarded to its second port even if it does not block at the first one.

The queuing value used to insert the message is zero for a Send operation or if the port discipline is FIFO. For a Surrogate_send to a priority port, the queuing value is the priority from the surrogate carrier.

A third operation, Cond send, never blocks. If the port is full, the message is not sent and a boolean parameter is cleared to false. Otherwise, the message is sent as described for Send, and the boolean parameter is set to true.

## MESSAGE RECEIVING

Receiving a message uses a carrier and two ports. The carrier receives the message at the first port and is then forwarded to the second port.

Two operations always receive messages. Receive implicitly uses the receiving process's carrier with the dispatching port as the second port. Surrogate receive uses an explicitly specified surrogate carrier and second port. For both operations:

1. If the port is empty, the carrier is appended to the FIFO queue of waiting carriers.

2. Otherwise, the first entry in the port message queue is dequeued. For Receive, the message is copied into the receiving context and the process carrier is never used. For Surrogate_receive, the message is copied into the surrogate carrier.

   If carriers are waiting (which now implies that the port is full), the first waiting carrier is dequeued, its message is inserted in the port message queue, and the dequeued carrier is forwarded to its second port.

   Last, for Surrogate__receive, the surrogate carrier is forwarded, carrying the received message to its second port. The receiving process must execute some form of receive on the second port to get the surrogate carrier, then invoke the Get carrier message operation to get the carried message.

A third operation, Cond_receive, never blocks. If the port is empty, no message is received and a boolean parameter is cleared to false. Otherwise, the message is received as described for Receive and the boolean parameter is set to true.

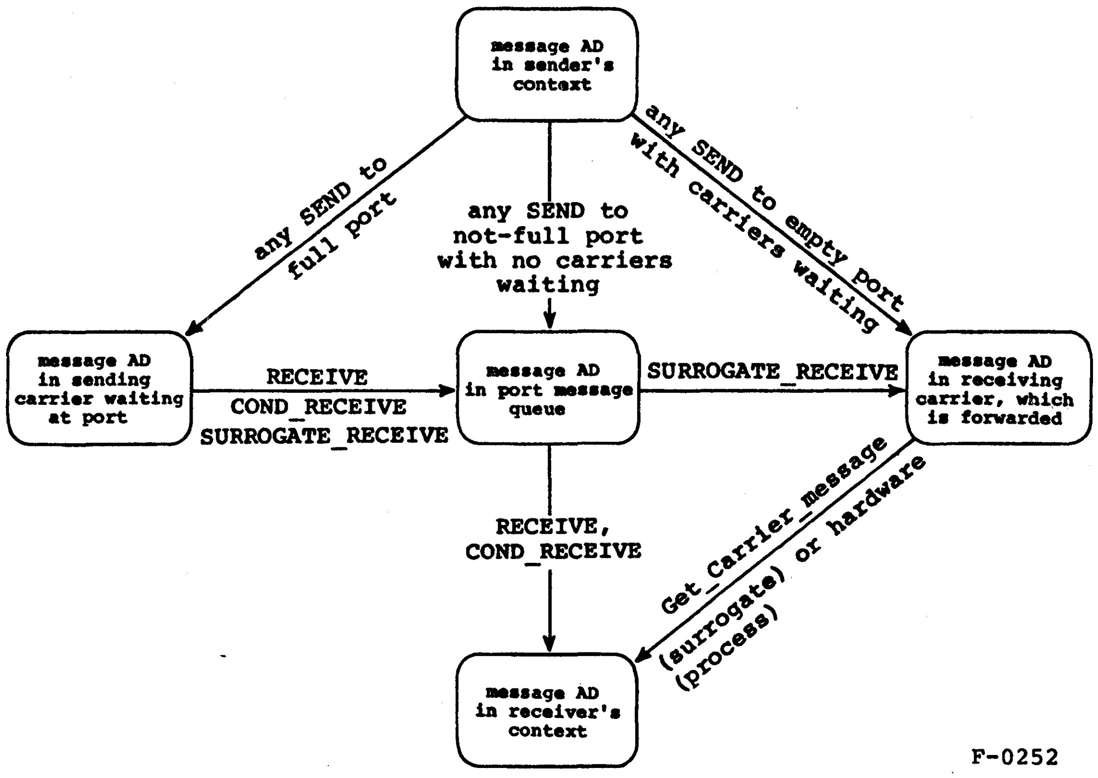

message AD in sender's context
any SEND to full port
any SEND to not-full port with no carriers waiting
any SEND to empty port with carriers waiting
message AD in sending carrier waiting at port
RECEIVE
COND_RECEIVE
SURROGATE_RECEIVE
message AD in port message queue
SURROGATE_RECEIVE
message AD in receiving carrier, which is forwarded
RECEIVE,
COND_RECEIVE
Get_Carrier_message (surrogate) or hardware (process)
message AD in receiver's context
F-0252

Figure COM-1. Message AD State Transitions

## CARRIER FORWARDING

When a surrogate send or receive completes, the surrogate carrier is sent to its second port. This second operation is called forwarding the carrier to its second port. The forwarding of a surrogate carrier is always optional; if the second port access in a surrogate operation is null, the carrier is not forwarded. The queuing value for the forwarding operation is always zero, regardless of the priority specified by the carrier for the primary operation. While the 432 architecture permits other queuing values for forwarding, they are not supported by iMAX. A receive operation on the second port returns an access for the carrier. A carrier that is forwarded from a Surrogate send operation carries no other message, and the Get_carrier_message operations returns null when applied to such a carrier. A carrier that is forwarded from a Surrogate_receive operation carries not only itself, but also the received message. The carrier must first be received from its second port, then the Get_carrier_mesage operation can be used to obtain an access for the carried message (and null it in the carrier).

Forwarding is also used to reschedule processes blocked in a simple Send or Receive operation. The carrier implicitly specified for these operations is the process carrier, and the implicit second port is the process's dispatching port. When a process that was waiting to receive a message is received by a processor from the dispatching port, the processor automatically completes the receive operation by copying the message access from the process carrier to the receiving context and then nulling the message access in the process carrier. This is the same operation that must be explicitly invoked for surrogate carriers as Get_carrier_message. A difference in the forwarding operation for the two types of carriers is that surrogate carriers are always forwarded (if their second port is not null) while process carriers are forwarded only if the operation blocks. This difference is understandable; if a process does not block, there is no need to reschedule it and thus no need to forward its carrier to the dispatching port.

## PACKAGE DESCRIPTIONS

The package Untyped_Ports defines the port operations described above, for messages of type any_access.

The package Typed_Ports contain three generic packages:

- Simple Port Def defines a new type of port that can only handle messages of an access type specified by the user. For this new type of port, Simple_Port_Def defines the operations Permits, Remove, Create_port, Send, Cond_send, Receive, and Cond_receive.

- Carrier Def defines a new type of carrier that can only carry messages of an access type specified by the user. For this new type of carrier, Carrier__Def defines the operations Permits, Remove, Create_carrier, Get_carrier_message, Get_carrier_id, and Set_carrier_priority. The Create_carrier function differs from the same function in the Untyped_Ports package: the caller no longer specifies the number of access slots in the visible carrier (but the number will always be one) but does supply an identification value of a type specified by the user (the identification type need not be an access type). The one access slot in the visible carrier references the identification value supplied when the carrier is created. The identification value for a carrier cannot be changed, and can be retrieved by the Get_carrier_id function.

- Surrogate Port Def defines surrogate operations for user-defined message types, port types, and carrier types. The user must specify a user_port type that can handle messages of the specified user_message type, a user_carrier type that can carry messages of the user_message type, and a user_carrier_port type that can handle messages of the user_carrier type. The following example shows how to do this.

## HOW TO USE THE GENERIC PACKAGES

This section gives an example of how to use the generic packages defined by the Typed_Ports package. For example, suppose you want to provide a complete set of port operations for messages that are integers:

```
type integer_message is access integer;

package Integer_Port_Def is
  new Typed_Ports.Simple_Port_Def(integer_message);
  --  Integer_Port_Def.user_port is an access type consisting of
  --  accesses for ports that can only handle messages of type
  --  integer_message.

package Integer_Carrier_Def is
  new Typed_Ports.Carrier_Def (
      user_message => integer_message;
      user_carrier_id => any_access);  --  ID can be any type
  --  Integer_Carrier_Def.user_carrier is an access type consisting
  --  of accesses for carriers that can only carry integer_messages.

package Integer_Carrier_Port_Def is
  new Typed_Ports.Simple_Port_Def(Integer_Carrier_Def.user_carrier);
  --  Integer_Carrier_Port_Def.user_port is an access type
  --  consisting of accesses for ports that can only handle carriers
  --  of integer_messages as messages.
```

```
package Integer_Surrogate_Port_Def is
  new Typed_Ports.Surrogate_Port_Def(
      user_port         => Integer_Port_Def.user_port;
      user_message      => integer_message;
      user_carrier      => Integer_Carrier_Def.user_carrier;
      user_carrier_port => Integer_Carrier_Port_Def.user_port);


  -- to make naming less cumbersome
  -- (in Ada, subtype is used to rename types):

subtype  integer_port is Integer_Port_Def.user_port;
subtype  integer_carrier is Integer_Carrier_Def.user_carrier;
subtype  integer_carrier_port is Integer_Carrier_Port_Def.user_port;


procedure Integer_port_example is

  -- Function
  --   Give an example of creating
  --   An integer port and sending
  --   a value (3) to it.

  example_port:  integer_port;
  example_message:  integer_message;

begin
  example_port := Integer_Port_Def.Create_port(10);
  example_message := new integer(3);
  Integer_Port_Def.Send (example_port, example_message);
end Integer_port_example;
```

## SURROGATE APPLICATIONS

This section describes two applications for surrogate carriers and the surrogate operations. A single procedure provided by the Surrogate Example package fills both application needs. The Surrogate_Example package is listed at the end of this chapter.

### PRIORITIZED MESSAGE SENDING

The Asynchronous send procedure can be used to send a message of specified priority to a port. The operation does not block the caller.

## SURE NOTIFICATION

A two-port protocol can enable a process to send a message to (notify) another process on which it must not depend; i.e., the first process must never block because the destination port is full. The Asynchronous_send procedure also provides this service. Asynchronous_send first obtains a carrier from a carrier_pool port, creating a new carrier if the pool is empty. The port and the carriers in it are supplied by the Surrogate_Example package and are invisible to the user. The carrier obtained is used to Surrogate_send the message to the notification port where the other process can receive it. Control returns immediately to the caller, without any possibility of blocking. If the send does block, it is the surrogate carrier and not the calling process that is blocked. After the send completes, the carrier is forwarded back to the carrier_pool port.

## UNTYPED PORTS PACKAGE

```
with Descriptor_Definitions, iMAX_Definitions;
package Untyped_Ports is

  -- Function:
  --   This package provides operations for the creation of ports and
  --   carriers, and provides the port operations.

  use iMAX_Definitions, Descriptor_Definitions;

  -- PORTS

  max_message_count:   constant := 8190;
    -- max number of messages in a port's message queue
    -- so the port fits in one segment

  send_rights:     constant rights := type_right_1;
  receive_rights:  constant rights := type_right_2;


  type q_discipline is (
      FIFO,        -- first_in_first_out
      priority);  -- within the same priority, FIFO is used


  function Create_port(
      message_count:   short_ordinal range 1 .. max_message_count;
        -- number of messages in the port's message queue
      port_discipline: q_discipline := FIFO;
        -- organization of the port's message queue
      sro:             storage_resource := null)
        -- SRO used in the creation
    return port;                         -- the created port

    -- Function:
    --   A port is created with the size of its message queue equal to
    --   message_count.  The discipline of the port's message queue can
    --   be optionally specified and defaults to FIFO.  The SRO used
    --   for the creation defaults to the default_global_heap_SRO of
    --   the calling process.

  procedure Send(
      prt:  port;          -- port to which a message is to be sent
      msg:  any_access);  -- message that is sent

    -- Function:
    --   The specified message is sent to the the specified port.
    --   If the message queue of the port is full, then the calling
    --   process blocks until a message slot becomes available.
```

```
procedure Receive(
    prt:  port;        -- port from which a message is to be received
    msg:  out any_access); -- received message

  -- Function:
  --   A message is received from the specified port.
  --   If no message is available, then the calling process
  --   blocks until a message becomes available.
  --   The received message is then returned to the caller.

procedure Cond_send(
    prt:      port;           -- port to which a message is to sent
    msg:      any_access;     -- message to be sent
    success:  out boolean);   -- true if message is in port, otherwise
                              -- false

  -- Function:
  --   An attempt is made to send the specified message to the
  --   specified port.  If the message queue of the port is full,
  --   then the message is not delivered and success is assigned
  --   false.  Otherwise,  the message is sent and success is
  --   assigned true.

procedure Cond_receive(
    prt:      port;     -- port from which a message is to be received
    msg:      out any_access;  -- received message, if any
    success:  out boolean);    -- true if message is received, false
                               -- otherwise

  -- Function:
  --   An attempt is made to receive a message from the specified
  --   port.  If the message queue of the port is empty, then success
  --   is assigned false and no message is received.  Otherwise, a
  --   message is received and success is assigned true.

-- CARRIERS

max_carrier_size:  constant := 16378;
  -- max number of slots in the refined carrier
  -- so the carrier fits in one segment

function Create_carrier(
    size: short_ordinal range 1 .. max_carrier_size := 1;
      -- # of slots in the carrier that are in the refined carrier
    pri:  short_ordinal := 0;  -- priority when this carrier is used
    sro:  storage_resource := null)   -- SRO used in the creation
  return carrier;                     -- created carrier

  -- Function:
  --   A carrier with the specified priority is created.  The sro
  --   used in the creation defaults to the calling process's default
  --   global heap sro.
  use_rights:  constant rights := type_right_1;


  procedure Get_carrier_message(
      car:  carrier;           -- carrier that has received a message
      msg:  out any_access);   -- message that the carrier has received

    -- Function:
    --   The message in the specified carrier is extracted from this
    --   carrier.  This operation then nulls the message access in the
    --   carrier.


  procedure Set_carrier_priority(
      car:  carrier;           -- carrier whose priority is to be changed
      pri:  short_ordinal);    -- new priority value

    -- Function:
    --   The priority of the specified carrier is set to the specified
    --   value.


  -- SURROGATE PORT OPERATIONS

  procedure Surrogate_send(
      prt:  port;         -- port to which message is to be sent
      msg:  any_access;   -- message to be sent
      car:  carrier;      -- carrier that is used in surrogate operation
      dst:  port);        -- second port where carrier is sent (as a
                          -- message) after the first send succeeds.

    -- Function:
    --   The specified message is sent to the specified port.  If the
    --   message queue of the port is full, then the carrier blocks
    --   until space becomes available in the message queue.  After
    --   the send succeeds, the carrier is forwarded to the specified
    --   second port as a message.


  procedure Surrogate_receive(
      prt:  port;      -- port from which a message is to be received
      car:  carrier;   -- carrier used to receive the message
      dst:  port);     -- second port where carrier is sent after
                       -- it receives a message

    -- Function:
    --   The specified carrier is used to receive a message from the
    --   specified port.  If there are no messages enqueued at the
    --   port, then the carrier blocks (waits) until messages become
    --   available.  When the receive succeeds, the carrier carrying the
    --   received message is forwarded as a message to the specified
    --   second port.

end Untyped_Ports;
```

## TYPED PORTS PACKAGE

```
with Descriptor_Definitions, iMAX_Definitions;
package Typed_Ports is

  -- Function:
  --   Typed_Ports consists of three packages which provide the user
  --   with a high level (Ada typed) view of ports, carriers and their
  --   operations.


  use iMAX_Definitions, Descriptor_Definitions;


  -- RIGHTS

  -- for ports
  send_rights:     constant rights := type_right_1;
  receive_rights:  constant rights := type_right_2;

  -- for carriers
  use_rights:      constant rights := type_right_1;

  generic
      type user_message is private;
        -- all messages that this package deals with are of this access
        -- type
  package Simple_Port_Def is

    -- Function:
    --   This package provides definitions and operations that enable
    --   the user to create ports, and do simple operations on those
    --   ports involving only messages of type "user_message".

    max_message_count:  constant short_ordinal := 8190;
      -- max number of messages in a port's message queue

    type user_port is private;     -- ports of this type can only be used
                                   -- with messages of type user_message

    null_user_port:  constant user_port;


    type q_discipline is (
        FIFO,        -- first_in_first_out, also the default q_discipline
        priority);   -- within same priority, FIFO is used
```

```
function Permits(
    prt:  user_port;
    r1:   rights;
    r2:   rights := no_such_right;
    r3:   rights := no_such_right;
    r4:   rights := no_such_right;
    r5:   rights := no_such_right;
    r6:   rights := no_such_right;
    r7:   rights := no_such_right)
  return boolean;

  -- Function:
  --   Check whether the given user_port has the specified rights.
  --   This function returns false for a null_user_port.


procedure Remove(
    prt:  in out user_port;
    r1:          rights;
    r2:          rights := no_such_right;
    r3:          rights := no_such_right;
    r4:          rights := no_such_right;
    r5:          rights := no_such_right;
    r6:          rights := no_such_right;
    r7:          rights := no_such_right);

  -- Function:
  --   The specified rights are removed from the given user_port.


function Create_port(
    message_count:    short_ordinal range 1 .. max_message_count;
      -- max number of messages in the port's message queue
    port_discipline:  q_discipline := FIFO;
      -- organization of the port's message queue
    sro:              storage_resource := null)
      -- SRO used in the creation
  return user_port;                -- user_port that is created

  -- Function:
  --   A user_port with the specified message_count and the
  --   specified message queue discipline is created.  The SRO used
  --   in the creation defaults to the default_global_heap_SRO.


procedure Send(
    prt:  user_port;       -- port to which a message is to be sent
    msg:  user_message);   -- message that is to be sent

  -- Function:
  --   The specified user_message is sent to the specified
  --   user_port.  If the send cannot succeed immediately, the
  --   calling process blocks.
```

```
    procedure Cond_send(
        prt:      user_port;    -- port to which a message is to be sent
        msg:      user_message;  -- message to be sent
        success:  out boolean);  -- true if send succeeded, false
                                 -- otherwise

      -- Function:
      --   An attempt is made to send the specified message to the
      --   specified port.  If the send cannot succeed immediately,
      --   then success is assigned false and the message is not sent.
      --   Otherwise, the message is sent and success is assigned true.


    procedure Receive(
        prt:  user_port;  -- port from which a message is to be received
        msg:  out user_message); -- received message

      -- Function:
      --   A message is received from the specified user_port.  The
      --   calling process blocks until the receive succeeds.


    procedure Cond_receive(
        prt:      user_port;          -- port from which a message is to
                                      -- be received
        msg:      out user_message;   -- received message, if any
        success:  out boolean);       -- true if message was received,
                                      -- false otherwise
      -- Function:
      --   An attempt is made to receive a message from the specified
      --   user_port.  If the receive cannot succeed immediately, then
      --   success is assigned false and no message is received.
      --   Otherwise, a message is received and success is assigned
      --   true.

  end Simple_Port_Def;
```

```
generic
    type user_message is private;
      -- type of message as specified by the user
    type user_carrier_id is private;
      -- type of carrier_id as specified by the user
package Carrier_Def is

  -- Function:
  --   Definitions and operations on carriers are provided in this
  --   package.


  type user_carrier is private;
    -- user_carriers can only carry messages of type user_message

  null_user_carrier:  constant user_carrier;


  function Permits(
      car:  user_carrier;
      r1:   rights;
      r2:   rights := no_such_right;
      r3:   rights := no_such_right;
      r4:   rights := no_such_right;
      r5:   rights := no_such_right;
      r6:   rights := no_such_right;
      r7:   rights := no_such_right)
    return boolean;

    -- Function:
    --   Checks whether the given user_carrier has the specified
    --   rights.  This function returns false for a null_user_carrier.


  procedure Remove(
      car:  in out user_carrier;
      r1:          rights;
      r2:          rights := no_such_right;
      r3:          rights := no_such_right;
      r4:          rights := no_such_right;
      r5:          rights := no_such_right;
      r6:          rights := no_such_right;
      r7:          rights := no_such_right);

    -- Function:
    --   The specified rights are removed from the given user_carrier.
```

```
   function Create_carrier(
      id:   user_carrier_id;           -- carrier will have this id
      pri:  short_ordinal := 0;        -- priority
      sro:  storage_resource := null)  -- SRO used for creation
     return user_carrier;              -- carrier that is created

     -- Function:
     --   A user_carrier with the specified id and priority is created.
     --   The SRO used for the creation defaults to the
     --   default_global_heap_SRO of the calling process.


  procedure Get_carrier_message(
      car:  user_carrier;        -- carrier from which we want to
                                 -- extract a message
      msg:  out user_message);   -- message previously received by the
                                 -- carrier

     -- Function:
     --   The message most recently received by the specified
     --   user_carrier is returned.  This operation then nulls the
     --   message access in the user_carrier.


   function Get_carrier_id(
      car:  user_carrier)      -- carrier whose id is requested
     return user_carrier_id;   -- id of the carrier

     -- Function:
     --   The id of the specified user_carrier is returned.


   procedure Set_carrier_priority(
      car:  user_carrier;   -- carrier whose priority is to be changed
      pri:  short_ordinal); -- new priority value

     -- Function:
     --   The priority of the specified user_carrier is set to the
     --   specified value.

  end Carrier_Def;
  generic
      type user_port is private;
        -- port capable of handling user_messages
      type user_message is private;
        -- type of messages as specified by the user
      type user_carrier is private;
        -- carrier capable of carrying user_messages
      type user_carrier_port is private;
        -- port capable of handling user_carriers
  package Surrogate_Port_Def is

    -- Function:
    --   This package contains surrogate port operations.

    -- Note:
    --   It is the programmers' reponsibility, when instantiating
    --   this package, to provide generic parameters that are
    --   in correct relation to one another.


    procedure Surrogate_send(
        prt:     user_port;
          -- port to which a user_message is to be sent
        msg:     user_message;     -- message that is to be sent
        car:     user_carrier;     -- carrier used in surrogate operation
        dst:     user_carrier_port);
          -- second port where carrier will be forwarded (carrying
          -- itself as a message) after the message is sent

      -- Function:
      --   The specified message is sent to the specified port.
      --   If the send cannot succeed immediately, then the specified
      --   carrier blocks.  When eventually the send succeeds, the
      --   carrier is forwarded to the specified second port.


      procedure Surrogate_receive(
        prt:     user_port;
          -- port from which a message is to be received
        car:     user_carrier;
          -- carrier used in surrogate operation
        dst:     user_carrier_port);
          -- second port where carrier will be forwarded (carrying
          -- itself as a message) after receiving a message

      -- Function:
      --   The specified carrier receives a message from the specified
      --   port.  If the receive cannot succeed immediately, the carrier
      --   blocks.  When eventually the receive succeeds, the carrier
      --   carrying the received message is forwarded to the specified
      --   second port.

  end Surrogate_Port_Def;
end Typed_Ports;
```

## SURROGATE EXAMPLE PACKAGE

```
with iMAX_Definitions;
package Surrogate_Example is

  --   Give example of using surrogate port operations.

  use iMAX_Definitions;

  procedure Asynchronous_Send(
      prt:  port;                -- port to which message is sent
      msg:  any_access;          -- message to be sent
      pri:  short_ordinal := 0);-- priority of message

    -- Function:
    --   Send a (prioritized) message in a carrier to a port on
    --   behalf of a process that cannot block or wait.

    --   The basic service provided by this function is to hide
    --   the details of creating and managing a pool of surrogate
    --   carriers.

    -- Note:
    --   If the message priority is used, the port should use the
    --   priority queuing discipline (or the message priority is
    --   ignored).

end Surrogate_Example;
```

SURROGATE EXAMPLE PACKAGE BODY

```
with iMAX_Definitions, Untyped_Ports;
package body Surrogate_Example is

  use iMAX_Definitions, Untyped_Ports;


  carrier_pool:  port := null;
  -- access for port used for queue of free carriers.
  -- The port is created on the first call to Asynchronous_send.

  procedure Asynchronous_Send(
      prt:  port;                 -- port to which message is sent
      msg:  any_access;           -- message to be sent
      pri:  short_ordinal := 0) -- priority of message
    is

     -- Logic:
    --    At the beginning of the procedure, the body must check if
    --    carrier_pool port exists yet.  If it does not, then it must
    --    be created.

    --    For each Asynchronous_send operation, the body tries to get
    --    a carrier from the pool, creating a new carrier only if the
    --    pool is empty.  The Surrogate_send operation forwards carriers
    --    back to the pool after they are used.  Note that there is no
    --    limit to the number of carriers that this procedure can create
    --    and add to the pool.

    my_carrier:  carrier;    -- carrier used for operation
    success:     boolean;    -- flag used for conditional receive.

  begin

    if carrier_pool = null then
      carrier_pool := Create_port(1);
    end if;

    Cond_receive(carrier_pool, my_carrier, success);
    if not success then
      my_carrier := Create_carrier(1);
    end if;

    Set_carrier_priority(my_carrier, pri);

    Surrogate_send(prt, msg, my_carrier, carrier_pool);

  end Asynchronous_send;

end Surrogate_Example;
```

# CHAPTER PEN
# PROGRAM ENVIRONMENT ACCESS

This chapter describes the iMAX interface to the 432 objects used to implement program environments:

- context objects, which represent particular calls to subprograms
- instruction objects, which contain executable machine instructions, typically for a single subprogram
- domain objects, which correspond to Ada packages and provide access to a collection of related subprograms, data structures, tasks, or other packages
- process globals access segment (PGAS), which provides access to process attributes, including fault information

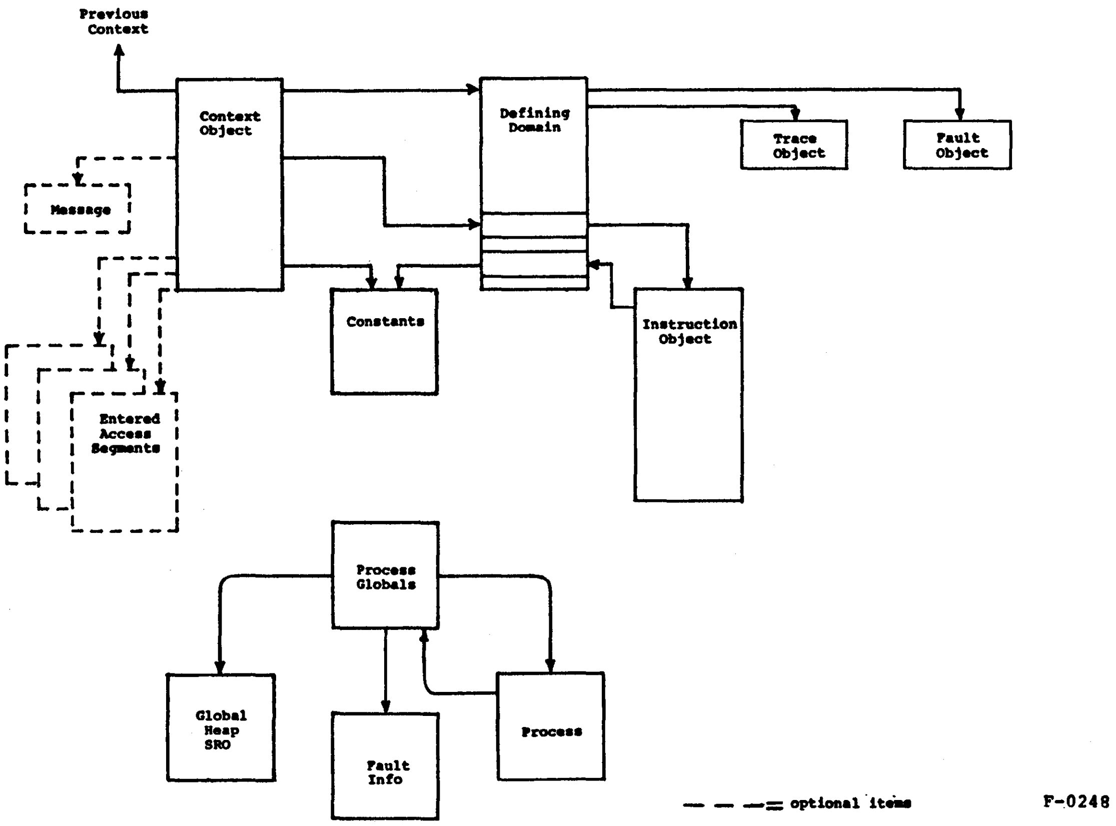


Figure PEN-1. Program Environment

Some of the facilities described are useful to the general user (e.g., Set context mode), but most are needed only by system implementers.

Figure PEN-1, Program Environment, illustrates the relationships between the objects that make up the program environment. The fault object and trace object are special instruction objects that cannot be called as procedures but act as handlers for context-level faults or trace events respectively.

The package Process Globals Definitions provides access to the process globals access segment. All the other facilities described are accessed through the Context Definitions package.

## CONTEXT ACCESS

Figure PEN-2 illustrates the context access environment which can be accessed using Context_Definitions. Context_Definitions provides the Current context function which returns an access for the current context access segment, represented as a context rep val record. The context access segment is the root of the context's access environment. The access returned by Current_ context has read and write rights, but no type rights. Table PEN-1 describes the accesses contained in the record and their rights. Note that delete rights are removed by the 432 GDP when entering an access segment or storing the access for the defining domain. However, the GDP ignores the absence of delete rights in an entered access segment slot when executing an ENTER ACCESS SEGMENT instruction to reference some other access segment.

The context data segment contains the context status, which indicates whether the context has faulted and gives the context mode fields that control how floating point computations are done within the context. The context data segment also contains the stack pointer (SP) and instruction pointer (IP) for the context. However, IP and SP are not defined while the context is active (they are cached on-chip and not updated until a new context is activated or until the process containing the context stops running). The remaining fields give the domain access index for the instruction object being executed, the domain access index for the trace instruction object, the trace instruction pointer, and the trace code.

The message field of the context access segment can reference parameters passed by the caller of the context, or a message received in interprocess communication. The Current context message function returns an AD for the current message object, or a null AD if there is no message.

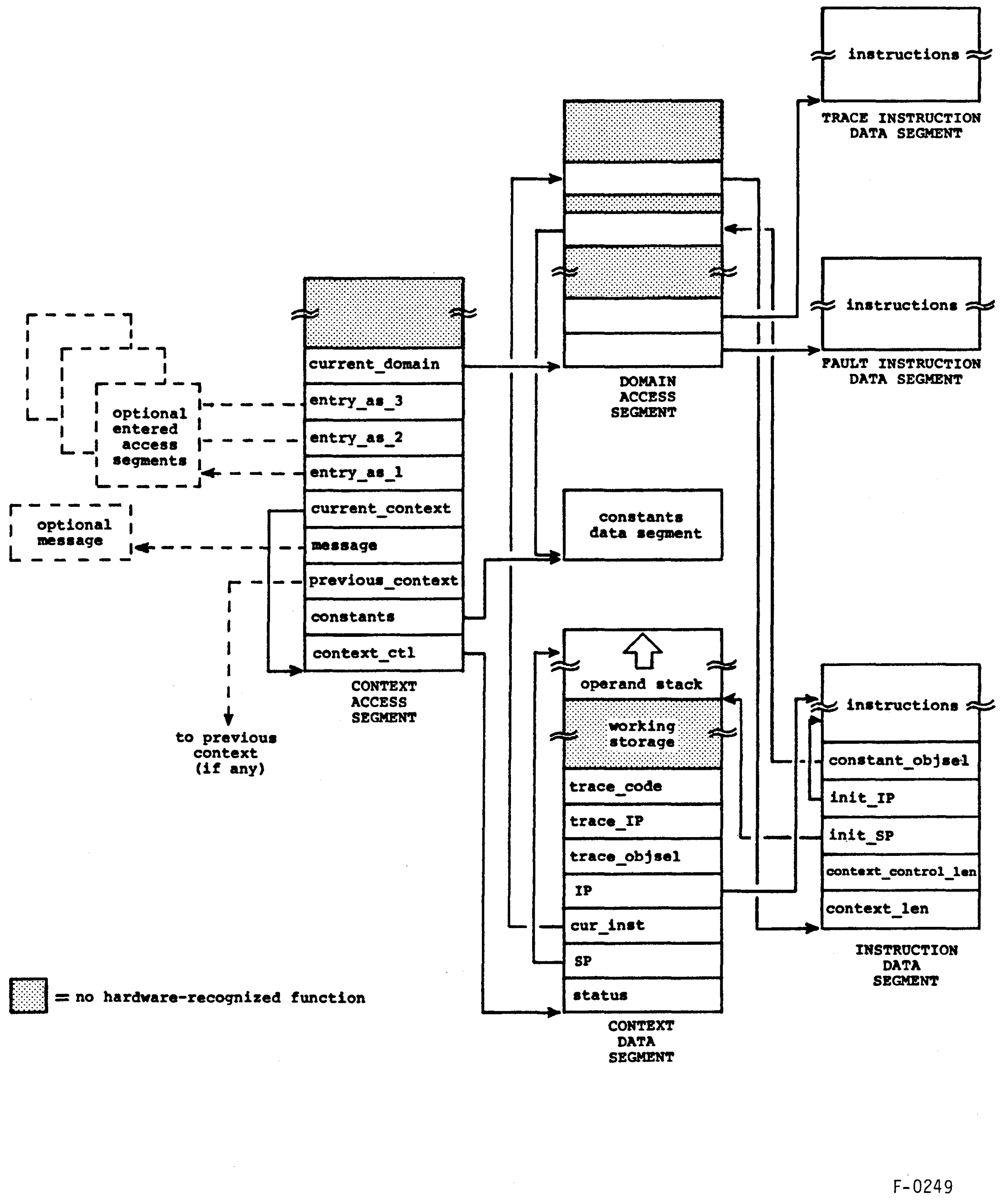


Figure PEN-2. Context Access Environment

Table PEN-1. Context Access Segment Fields and Rights

| Name | Description | Rights | | |
|---|---|---|---|---|
| | | READ | WRITE | DELETE |
| context_ctl | references context data segment | YES | YES | NO |
| constants | references a data segment containing constants used by this context | YES | NO | NO |
| previous_context* | references previous context's context access segment (null only if this is first context of a process) | NO | NO | NO |
| message* | references optional context parameters or message received in interprocess communication | ** | ** | ** |
| current_context | references this segment (context access segment) which is "entered access segment 0" | YES | YES | NO |
| entry_as_1.. entry_as_3* | references optional additional entered access segments which reference objects directly addressable for this context | ** | ** | NO |
| current_domain | references domain through which this context was called | YES | NO | NO |

*Optional items can be null

**User-supplied access has user-supplied rights

## CONTEXT MODE CONTROL

Three fields in a context's status control how floating-point computations within the context are done by the GDP.

If the boolean field fault on inexact is true, the context faults whenever a floating point operation generates a result that cannot be exactly represented in the destination format. For example:

```
        C := A/B;       -- where A, B, and C are reals, and
                        -- fault_on_inexact is true.
```

If A is 5.0 and B is 3.0, then the computation will fault when it is attempted, because the true result of 1.6666... cannot be exactly represented as a binary floating point number. If fault__on__inexact is false, then the computation proceeds, with an approximate result. One application of the fault_on_inexact field is the simulation of 64-bit integer arithmetic using the 432 temporary-real data type.

The precision control field restricts the precision of floating point computation but does not affect ordinal or integer computation. Its values are defined by the precision enumeration. If precision_control is short_real, then all real or temporary_real operands and results are rounded to 24 bits after the binary point. For real operands and results, only 23 of the 24 bits appear in the significand, because real values are normalized and the significand has an implied leading bit that is always one. For temporary_real operands and results, all 24 bits appear in the significand. If precision_control is real, then all temporary__real operands and results are rounded to 53 bits in the significand. If precision_control is temporary_real, then the full precision available in all three formats is used. Precision_control does not limit the exponent range of any of the formats, but limits only the use of the significands. Precision_control is mandated by the proposed IEEE floating point standard for use in running programs which must be compatible with the limited precision available on some machines. Varying the precision of a computation is also useful for determining the effect of changed precision within a computation on the accuracy of its results. Users will normally want full temporary-real precision.

The rounding control field controls the rounding of the results of computational instructions when the internal result cannot be exactly represented in the destination format. Note that rounding_control is only useful if fault_on_inexact is false. The values that the field can have are defined by the rounding enumeration. An internal result which requires rounding is always between two numbers that can be represented in the destination format. The rounding_control field determines which of the two numbers is chosen as the rounded result. The round even mode normally rounds to the nearer of the two numbers; if the internal result is equi-distant from both numbers, then the one with a last bit that is zero is chosen. The round up mode always rounds to the greater of the two numbers. Round down always rounds to the lesser of the two numbers. Truncate always rounds to the number that is closest to zero (the lesser if the result is positive, and the greater if the result is negative).

The Set context mode procedure assigns all three fields when it is called. The default context mode values for both iMAX and the Set context mode procedure are:

```
fault_on_inexact  := false;
precision_control := temporary_real;
rounding_control  := round_even;
```

The default context mode values are used by iMAX and Ada unless the user changes them.

When a subprogram is called, the new context inherits its mode values from its caller.

The Context Access section (above) describes how to read the mode values.

**NOTE**

Directly writing the status field of the record context control rep val will not alter the mode bits used by the processor because the processor caches this information. Only the Set__context__mode procedure or the equivalent SET MODE GDP operator should be used to modify the mode bits. Both the Ada procedure and the GDP operator will change the mode in both the processor and the status field.

The GDP's support for floating point computations is described in more detail in the iAPX 432 General Data Processor Architecture Reference Manual.

## INSTRUCTION SEGMENT ACCESS

A user may want to access an instruction segment:

- to determine the size of the context created by a call to the instruction segment
- to find the constants object for the segment (given the defining domain)
- to retrieve and decode parts of the instruction stream for fault-handling

iMAX defines the access type instruction segment rep which references an instruction segment rep val record. Its components are:

- size in bytes (minus one) of the context access segment and context data segment required by the instruction segment
- initial instruction pointer value -- the entry point for this instruction segment
- initial stack pointer value -- byte offset of the start of the operand stack in the context data segment
- domain access indices for the constants object associated with this instruction segment. All instruction segments have an associated constants object.

The instructions are not represented in the record. However, user software can read instruction streams from Ada by retyping the instruction object access to a user-defined type that views the instruction object as a packed bit array, and then reading that array.

## DOMAIN OPERATIONS

The 432 architecture provides a run-time representation for Ada packages as domain objects. A domain object is an access segment that contains accesses for all the data elements or operations defined by the package it represents. The public part of the package can then be represented by a refinement of the domain.

Context Definitions provides operations that create and refine domains, using either a heap SRO or the process stack SRO:

```
Create_domain
Create_stack_domain
Create_domain refinement
Create_stack_domain_refinement
```

A "stack domain" is simply a domain created from the process stack SRO, and a "stack domain refinement" is simply a domain refinement created from the process stack SRO. Note that a stack domain refinement can be created for a domain that is itself created using a heap SRO. The access returned from a created domain has all access rights. The access returned for a refined domain has whatever access rights are present on the access parameter that references the domain being refined.

Users of the operations that create domains are responsible for establishing the processor-recognized fields in the created domains, including accesses for a fault object and a trace object. ADs for any constants objects in the domain must be in the positions designated by whatever instruction objects are accessed through the domain.

Context_Definitions also provides the Retype to domain rep function which takes a domain access and returns the exact same access, but now referencing an any_access_array_val. This retyping enables the user to manipulate the domain as an array of any_access values.

## PROCESS GLOBALS ACCESS SEGMENT (PGAS)

iMAX associates with each process an access segment with the processor-recognized function of providing a process-wide access environment available to all contexts created by a particular process. Any context can access the globals access segment by executing the ENTER GLOBAL ACCESS SEGMENT instruction. Using Ada, a context can invoke the function Process globals, which returns an access for the PGAS with read and write rights; the fields of the PGAS are defined by the record type process globals rec. The Process Globals Definitions package also provides the Retype to process globals rep function which allows an any_access value to be viewed as an access to a process globals_rec record.

The PGAS provides a dynamic linking mechanism by providing accesses to instances of needed package types; the instances are represented by 432 domain objects. All are provided for use by the Ada compiler. Other PGAS fields provided for the compiler are:

- access to a type control object (TCO) used to create domains
- access to a refinement control object (RCO) used to refine domains
- access to a global heap SRO
- access to the context fault area for the process
- access to a list of open I/O files, including standard input, output, and error files

Table PEN-2 lists process globals fields and rights.

Table PEN-2. Process Globals Fields and Rights

| Name | Description | Rights | | |
|---|---|---|---|---|
| | | READ | WRITE | DELETE |
| owner | process that own this PGAS | NO | NO | NO |
| domain_type_control | used by compiler-produced code that creates domain | NO | NO | NO |
| default_global_heap_SRO | used as default for creating heap objects | NO | NO | NO |
| domain_refinement_control | used by compiler-produced code that refines domains | NO | NO | NO |
| context_fault_area | context fault area for process | YES | NO | NO |
| debug_status | used by DEBUG-432 debugger | NO | NO | NO |
| sro_manager | for compiler, access to package that creates local heaps | NO | NO | NO |
| extended_type_manager | for compiler, access to package that creates type definition objects | NO | NO | NO |
| simulator | for compiler | NO | NO | NO |
| process_manager | access to manager package responsible for process (null until Ada tasking is implemented). | NO | NO | YES |
| open_file_list | list of all open files including standard files | YES | YES | NO |
| runtime_environment_list | list of language-specific run-time environments | YES | YES | NO |

In iMAX V2, the process_manager entry is null. Note that the entries in the open_file_list and runtime_environment_list can be modified by the user.

In future releases, a PGAS may contain other iMAX-defined items; therefore, the first 20 accesses in each PGAS are reserved by Intel, though only the first 12 accesses are now defined. Users extending the PGAS should use a PGAS size of at least 20 + n accesses (80 + 4n bytes), where n is the number of user-defined accesses at the end of the PGAS.

## FAULT INFORMATION

The context fault area contents defined in Context_Definitions are accessed via Process_Globals_Definitions. Table PEN-3 describes the different fields of the fault area, but does not document hardware fault conditions, the circumstances in which they occur, or the codes that are stored as a result. The reader should refer to the *iAPX 432 General Data Processor Architecture Reference Manual* for this information.

Table PEN-3. Fault_info_rec Fields

| | |
|---|---|
| flt_acc_index | access index into defining domain of AD for faulted instruction object |
| post_inst_IP | instruction pointer after the instruction that faulted |
| pre_inst_IP | instruction pointer of the instruction that faulted |
| post_inst_SP | stack pointer after the instruction that faulted |
| pre_inst_SP | stack pointer before the instruction that faulted |
| flt_status | |
| result_destin | true if result was stored to destination |
| inexact | true if result was inexact (required rounding) |
| pre-inst_stack full | true if the top double-byte of the stack was on-chip before the instruction that faulted |
| post_inst_stack_full | true if the top double-byte of the stack was on-chip after the instruction that faulted |
| execution_state | an indicator of which part of a high-level instruction (e.g., SEND) was being executed when the fault occurred. |
| prcs_status | process status when the fault occurred. |
| psor_status | processor status when the fault occurred. |
| operator_id | operator ID of the instruction that faulted. |
| fault_handled | set to true when fault has been handled. |
| fault_code | indicates cause of fault |
| fault_acc_sel | access selector component of data reference that caused fault |
| fault_disp | displacement component of data reference that caused fault |
| second_source_or_exc_r | second source operand or exact result |
| first_source | first source operand |

All of these fields default to zero (booleans false, access indices null).

CONTEXT DEFINITIONS PACKAGE

```
with iMAX_Definitions, Descriptor_Definitions;
package Context_Definitions is

  -- Function:
  --   In this package software structures are given which map into the
  --   objects associated with a context.  The package  is available
  --   widely within iMAX and to users. It can be invoked from a low
  --   level, therefore its contents must be frozen.
  --   The function Current_context returns a reference to the current
  --   context of the caller.  The function Current_context_message
  --   returns a reference to the message carried by the current
  --   context.  Subprograms for creating domains and domain
  --   refinements are also made available.


  use iMAX_Definitions;

  -- TYPES NEEDED FOR CONTEXT FAULT INFORMATION

  -- (Types given here - information in process object and accessed
  -- using Process_Globals_Definitions.fault_info)

  type execution_states is (phase0, phase1, phase2, phase3,
                            phase4, phase5, phase6, phase7);

  type fault_status_rec is
    record
      result_destin:        boolean := false;  -- true if result stored
      inexact:              boolean := false;  -- true if result inexact
      pre_inst_stack_full:  boolean := false;  -- true if top of stack
                                               -- on chip before
                                               -- faulting instruction
      post_inst_stack_full: boolean := false;  -- true if top of stack
                                               -- on chip when fault
                                               -- occurs
      execution_state:      execution_states := phase0;
                                               -- indicates stage in
                                               -- instruction execution
                                               -- where fault occurred
    end record;
```

```
type fault_info_rec is
    -- This is the processor/process/context fault information
    -- in the process data segment.
  record
    flt_acc_idx:            access_selector  := null_access_selector;
      -- access selector for instruction segment which faulted
    post_inst_IP:           short_ordinal    := 0;
      -- instruction pointer when fault occurs
    pre_inst_IP:            short_ordinal    := 0;
      -- instruction pointer before faulting instruction
    post_inst_SP:           short_ordinal    := 0;
      -- stack pointer when fault occurs
    pre_inst_SP:            short_ordinal    := 0;
      -- stack pointer before faulting instruction
    flt_status:             fault_status_rec;
      -- gets default component values
    prcs_status:            short_ordinal    := 0;
    psor_status:            short_ordinal    := 0;
    operator_id:            short_ordinal range 0 .. 2**14    := 0;
    fault_handled:          boolean          := false;
    fault_code:             short_ordinal    := 0;
    fault_acc_sel:          short_ordinal    := 0;
    fault_disp:             short_ordinal    := 0;
    second_source_or_exc_r: temporary_real   := 0.0;
      -- exact result or second operand value
    first_source:           temporary_real   := 0.0;
      -- first operand value
  end record;

-- TYPES NEEDED FOR CONTEXT CONTROL(DATA) SEGMENT

type precision is (temporary_real, real, short_real);

type rounding is (round_even, round_up, round_down, truncate);

type context_status is
  record
    faulted:                boolean;    -- fault has occurred
    fault_on_inexact:       boolean;    -- context is to fault on
                                        -- inexact result
    precision_control:      precision;  -- determine representation
                                        -- of intermediates
    rounding_control:       rounding;   -- direction of rounding or
                                        -- truncate
  end record;
```

```
type context_control_rep_val is              -- data segment
  record
    status:                  context_status; -- see type declaration
                                             -- above
    SP:                      short_ordinal;  -- stack pointer
    cur_inst:                access_selector;-- offset in defining
                                             -- domain
    IP:                      short_ordinal;
    trace_accsel:            access_selector;-- offset in defining
                                             -- domain
    trace_IP:                short_ordinal;
    trace_code:              short_ordinal;
  end record;

type context_control_rep is access context_control_rep_val;

-- TYPES NEEDED FOR CONTEXT (ACCESS) SEGMENT


type context_rep_val is      -- access segment
  record
    context_ctl:             context_control_rep;
      -- context data segment
    constants:               data_sgt;
      -- constant data segment(null if none)
    previous_context:        context;
      -- caller(null if none)
    message:                 any_access;
      -- subprogram parms(null if none)
    current_context:         access_sgt;
      -- entered access segment 0(this segment)
    entry_as_1:              access_sgt;
      -- entered access segment 1(null if none)
    entry_as_2:              access_sgt;
      -- entered access segment 2(null if none)
    entry_as_3:              access_sgt;
      -- entered access segment 3(null if none)
    current_domain:          domain;
      -- defining domain
  end record;

type context_rep is access context_rep_val;
```

```
-- TYPES NEEDED FOR INSTRUCTION SEGMENT

type instruction_segment_rep_val is
  record
    context_len:              short_ordinal;
      -- context access segment length(bytes) - 1
    context_control_len:      short_ordinal;
      -- context data segment length(bytes) - 1
    init_SP:                  short_ordinal;
      -- context initial stack pointer
    init_IP:                  short_ordinal;
      -- context initial instruction pointer
    constant_accsel:          access_selector;
      -- offset into defining domain of access for data constants
      -- object
  end record;

type instruction_segment_rep is access instruction_segment_rep_val;

trace_rights: constant Descriptor_Definitions.rights :=
    Descriptor_Definitions.type_right_2;


-- TYPES NEEDED FOR DOMAIN

subtype domain_rep_val is any_access_array_val;
-- Zeroth entry references fault instruction segment.
-- First entry references trace instruction segment.

type domain_rep is access domain_rep_val;

-- REQUIRED MAINLY BY MEMORY MANAGEMENT:


function Current_context
  return context_rep;     -- a context object

  -- Function:
  --   Returns a reference for the context within which the function
  --   Current_context was called.
-- AVAILABLE TO ANY USER TO MODIFY STATUS OF HIS CURRENT CONTEXT:

procedure Set_context_mode(
    fault_on_inexact:  boolean := false;
    precision_control: precision := temporary_real;
    rounding_control:  rounding := round_even);

  -- Function:
  --   Modifies mode settings for the calling context, both in the
  --   processor and in the context status field of the calling
  --   context's data segment.

-- REQUIRED MAINLY BY PROCESS MANAGEMENT:

function Current_context_message
  return any_access;

  -- Function:
  --   Returns an access for the calling context's message.

-- OPERATIONS CREATING AND REFINING DOMAINS

function Retype_to_domain_rep(
    dmn: domain)
  return domain_rep;

  -- Function:
  --   Gives caller a representation he can handle.


procedure Create_domain(
    length:      short_ordinal;  -- length of domain(bytes) - 1;
    dmn:     out domain_rep;     -- domain created
    SRO:         storage_resource := null);
                                 -- SRO for create

  -- Function:
  --   A heap domain of the specified size is created.
  --   If the SRO parameter is defaulted, then the default global
  --   heap SRO in process globals is used for the create.
  --   User must ensure that first two entries reference fault
  --   and trace instruction objects respectively.
  procedure Create_domain_refinement(
      dmn:          domain_rep;         -- access for domain to be refined
      offset:       short_ordinal;      -- offset of the refinement(bytes)
      length:       short_ordinal;      -- length of the refinement(bytes)
                                        -- minus one
      rfn_dmn: out domain_rep;          -- access for the resulting
                                        -- refinement
      SRO:          storage_resource := null);
                                        -- SRO for create

    -- Function:
    --    A heap refinement is created from the specified domain,
    --    beginning at offset (bytes) and length (bytes) long.  The
    --    returned access has same rights as those on access passed in
    --    for the defining domain.  User must ensure that first two
    --    entries reference fault and trace instruction objects
    --    respectively.

  procedure Create_stack_domain(
      length:     short_ordinal;  -- length of domain(bytes) - 1
      dmn:    out domain_rep);    -- domain created

    -- Function:
    --    A stack domain of the specified size is created.
    --    User must ensure that first two entries reference fault
    --    and trace instruction objects respectively.


  procedure Create_stack_domain_refinement(
      dmn:          domain_rep;         -- access for domain to be refined
      offset:       short_ordinal;      -- offset of the refinement(bytes)
      length:       short_ordinal;      -- length of the refinement(bytes)
                                        -- minus one
      rfn_dmn: out domain_rep);  -- access for the resulting refinement

    -- Function:
    --    A stack refinement is created from the specified domain,
    --    beginning at "offset" (bytes) and "length" (bytes) long.
    --    The returned access has same rights as those on access passed
    --    in for the defining domain.
    --    User must ensure that first two entries reference fault
    --    and trace instruction objects respectively.


end Context_Definitions;
```

## PROCESS GLOBALS DEFINITIONS PACKAGE

```
with iMAX_Definitions, Context_Definitions, Unchecked_Conversion,
     Synchronous_IO_Interfaces;
package Process_Globals_Definitions is

  -- Function:
  --   This package defines types and procedures for handling
  --   process globals access segments.  This module defines the first
  --   12 slots of a process globals access segment (PGAS).
  --   The first 20 slots of a PGAS are reserved by Intel for future
  --   extension and specification.  Extended PGAS definitions should
  --   be consistent with the Intel specification.


  use iMAX_Definitions;

  type fault_info is access Context_Definitions.fault_info_rec;

  package type Simple_PM is

    -- This package type will be defined when Ada tasking is supported
    -- by the 432 Ada compiler.
  end Simple_PM;

  package type Simulated_Instructions is

    procedure Return_and_fault;

      -- Function:
      --   This procedure will simulate a return_and_fault instruction.
      --   For example, suppose A calls B which calls this procedure.
      --   The state of A's context will be changed so that when B
      --   actually returns to A, A's context-level fault handler
      --   will be invoked.  The Ada compiler uses this procedure
      --   to implement exception handling.
  end Simulated_Instructions;
```

```
package type Extended_Type_Management is

  function Create_type_definition(
     n:    print_name;         -- only attribute of type definition
     sro:  storage_resource := null)   -- SRO used in creation
    return type_definition;     -- created type definition object

    -- Function:
    --   A type_definition object is created with the specified
    --   print_name as only attribute.  The object is created
    --   from the specified SRO which defaults to the
    --   default_global_heap_sro.
end Extended_Type_Management;

package type SRO_Management is

  function Create_local_heap(
     relative_level:        short_ordinal := 0) -- relative level
                                                -- number.
    return storage_resource;

    -- Function:
    --   This function creates a local heap SRO.
    --   The parameter determines the lifetime of the local heap SRO
    --   and the objects created from it.  Specifically, the parameter
    --   selects a context object (i.e. its lifetime).  For example,
    --   if A calls B calls C which calls this function passing 0,
    --   the lifetime of the created local heap SRO will be tied
    --   to C's context; if 1 is passed, B's context; if 2, A's, etc.
    --   If the parameter exceeds the depth of the process's Context
    --   stack, a "level_error" exception is raised.
end SRO_Management;


-- The open file list includes three predefined entries for the
-- standard files which are open when the process begins execution.

type open_file_list_rep_val is
  record
    standard_input:  Synchronous_IO_Interfaces.Source;
    standard_output: Synchronous_IO_Interfaces.Sink;
    standard_error:  Synchronous_IO_Interfaces.Sink;
  end record;

type open_file_list_rep is access open_file_list_rep_val;
```

```
-- The runtime environment list is intended to allow addition of
-- environments for languages other than Ada in the future as they
-- are introduced.

type runtime_list_rep_val is
  record
    Ada:       any_access;
  end record;

type runtime_list_rep is access runtime_list_rep_val;



-- The definition of a PGAS.  All the access descriptors in a PGAS
-- lack delete rights with the exception of "process_manager".

type process_globals_rec is
  record
    owner:                      process;       -- BPM process object
                                               -- that owns PGAS.
    domain_type_control:        type_control;  -- used by compiler-
                                               -- produced code that
                                               -- creates a domain.
    default_global_heap_SRO:    storage_resource; -- used as default
                                               -- for heap allocations
                                               -- of permanent objects.
    domain_refinement_control: refinement_control; -- used by
                                               -- compiler-produced
                                               -- code that refines a
                                               -- domain.
    context_fault_area:         fault_info;    -- refinement of context
                                               -- fault area in process
                                               -- object.
    debug_status:               any_access;    -- for debugger.
    sro_manager:                SRO_Management; -- creates local heaps.
    extended_type_manager:      Extended_Type_Management;
                                               -- creates type
                                               -- definition objects.
    simulator:                  Simulated_Instructions;
                                               -- simulates a
                                               -- Return_and_Fault.
    process_manager:            Simple_PM;     -- manager responsible
                                               -- for process owning
                                               -- this PGAS.
    open_file_list:             open_file_list_rep; -- list of all
                                               -- open files including
                                               -- standard files
    runtime_environment_list:  runtime_list_rep; -- list of language-
                                               -- specific runtime
                                               -- environment info.
  end record;
  type process_globals_rep is access process_globals_rec;

  function Process_globals
    return process_globals_rep;      -- running process's PGAS.

    -- Function:
    --   The process globals access segment of the caller's process
    --   is returned.


  function Retype_to_process_globals_rep is
      new Unchecked_Conversion(any_access, process_globals_rep);


end Process_Globals_Definitions;
```

## PROGRAM ENVIRONMENT EXAMPLES PACKAGE

```
with iMAX_Definitions;
package Program_Environment_Examples is

  -- Function:
  --   Give examples of program environment access:
  --   1) Get level number of current context.
  --   2) Get number of bytes required for the context created
  --      by a subprogram call.

  use iMAX_Definitions;

  function Current_level
    return short_ordinal;

    -- Function:
    --   Return the level number of the calling context.


  function Context_size(
      p: instruction_segment)
    return ordinal;

    -- Function:
    --   Given an access for a subprogram, return the number of bytes
    --   required by the context object created when the subprogram is
    --   called.

    --   The size returned includes the segment header, pad bytes (if
    --   any) and object descriptor required for each segment.


private

  pragma inline(Current_level);
  -- required for correct operation; not just an optimization!

end Program_Environment_Examples;
```

PROGRAM ENVIRONMENT EXAMPLES PACKAGE BODY

```
with iMAX_Definitions, Descriptor_Definitions,
     Context_Definitions, Unchecked_Conversion;
package body Program_Environment_Examples is

  use iMAX_Definitions, Descriptor_Definitions, Context_Definitions;

  -- used by Context_size function
  function Retype_to_instruction_segment_rep is
    new Unchecked_Conversion(any_access, instruction_segment_rep);


  function Current_level
    return short_ordinal
  is

  begin
    return Inspect_object(any_access(Current_context())).od.level;
  end Current_level;


  function Real_size(     -- used by Context_size function
      len: short_ordinal)
    return ordinal
  is

    -- Logic:
    --   The numerical magic ((L + 8) / 8) * 8  is used to round
    --   the segment length up to a multiple of 8.  8 is added to
    --   L rather than 7 because L is really length - 1.  For example,
    --   if the segment's apparent size is 9 bytes, its associated
    --   length field is 8, and the numerical magic correctly yields
    --   16 (9 visible bytes plus 7 pad bytes).  The last term in
    --   the return statement then adds in the 8 bytes for the
    --   segment header and returns a result of 24.

  begin
    return ((ordinal(len) + 8) / 8) * 8 + 8;
  end Real_size;
  function Context_size(
    p: instruction_segment)
    return ordinal
  is

    i: instruction_segment_rep
         := Retype_to_instruction_segment_rep(p);

  begin
    return Real_size(i.context_len) +
           Real_size(i.context_control_len) +
           2 * OT_entry_sz;
  end Context_size;

end Program_Environment_Examples;
```

# CHAPTER EXT
# EXTENDED TYPES

This chapter describes how the iMAX package Extended_Type_Manager manages the creation of new extended types, the application of extended types to objects, and access to extended-type objects.

Extended typing provides two basic capabilities:

- to maintain software-defined types at run-time, so that an extended-type object provides access both to an object being typed and to a second object that represents the type of the first object.

- to control access to extended-type objects

An extended-type object can be manipulated like any other object via access descriptors that reference it. For example, an AD for an extended-type object can be transferred as a message between processes. Figure EXT-1 shows the structure of an extended-type object.

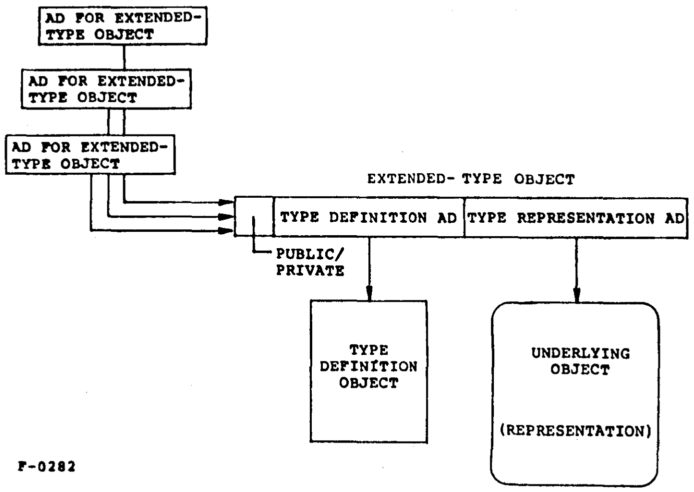


Figure EXT-1. Extended-Typing

The extended-type object associates a type definition object (TDO) with the underlying object being typed. The TDO acts as a type label for the object. For example, a module with a parameter of an extended type can check the type label to ensure that it is correct.

The underlying object in an extended-type object cannot be read or written with an AD for the extended-type object. Instead, the holder of an AD for the extended-type object must execute an explicit retrieve operation to get an AD for the underlying object. The underlying object is called the representation of the extended-type object.

The right to retrieve the representation of an extended-type object can be controlled using the public or private attribute of the object. This attribute is specified when the extended-type object is created and cannot be changed. Any module with an access for a public extended-type object can obtain an access for the type representation. The capability gained by using a public extended-type object is solely to be able to ask "What is the type of this object?" By contrast, to obtain an access for the representation of a private extended-type object, a module must present an access for the type definition of the object, and the access presented must have retrieve_rights (described below). Thus a type manager can distribute accesses for an extended-type object and be certain that no module obtaining such an access can operate on the underlying object -- so long as the type manager does not distribute type definition accesses with retrieve rights. An example of such a type manager, the Stack Manager package, appears at the end of this chapter.

Extended types defined using Extended_Type_Manager are independent of both Ada types and 432 object types. Objects that have the same extended type can have different Ada or object types. Objects that have different Ada or object types can have the same extended type. Extended types can also be layered, so that an extended type is applied to an object that is itself an extended-type object.

## TYPE DEFINITION OBJECTS

Type Definition Objects (TDOs) are a processor-recognized access segment type. In its most general form, a TDO is a collection of objects that defines the semantics or any other attributes that a user wishes to associate with a type. In iMAX V2, the form and use of the TDO are fixed. Each TDO has only a single attribute, of type print_name.

A new extended-type (a new TDO) is created by the Create Type Definition function, specifying a print_name and heap SRO. The heap SRO defaults to the default_global_heap SRO of the calling process. iMAX does not currently support the creation of TDOs from the stack SRO, though the architecture allows it. The returned access has all system rights and no representation rights.

TDO system rights are create rights and retrieve rights. Create rights are required to create an extended-type object of the type (to create an instance of the type). Retrieve rights are required to retrieve the representation of a private extended-type object.

## CREATING EXTENDED-TYPE OBJECTS

A new instance of an extended type is created by the Create public or Create private functions, using a specified type representation, type definition, and heap SRO. The returned access has all access rights. The type representation access can be null. The type definition access must have create rights and thus cannot be null. If this is not the case, the constraint_error exception is raised. The heap SRO defaults to the default_global_heap_SRO of the calling process. The lifetime of the heap SRO used (and thus of the new object) must be less than or equal to both the lifetime of the type representation (if non-null) and the lifetime of the type definition. If this is not the case, constraint_error is raised.

## ACCESS TO EXTENDED-TYPE OBJECTS

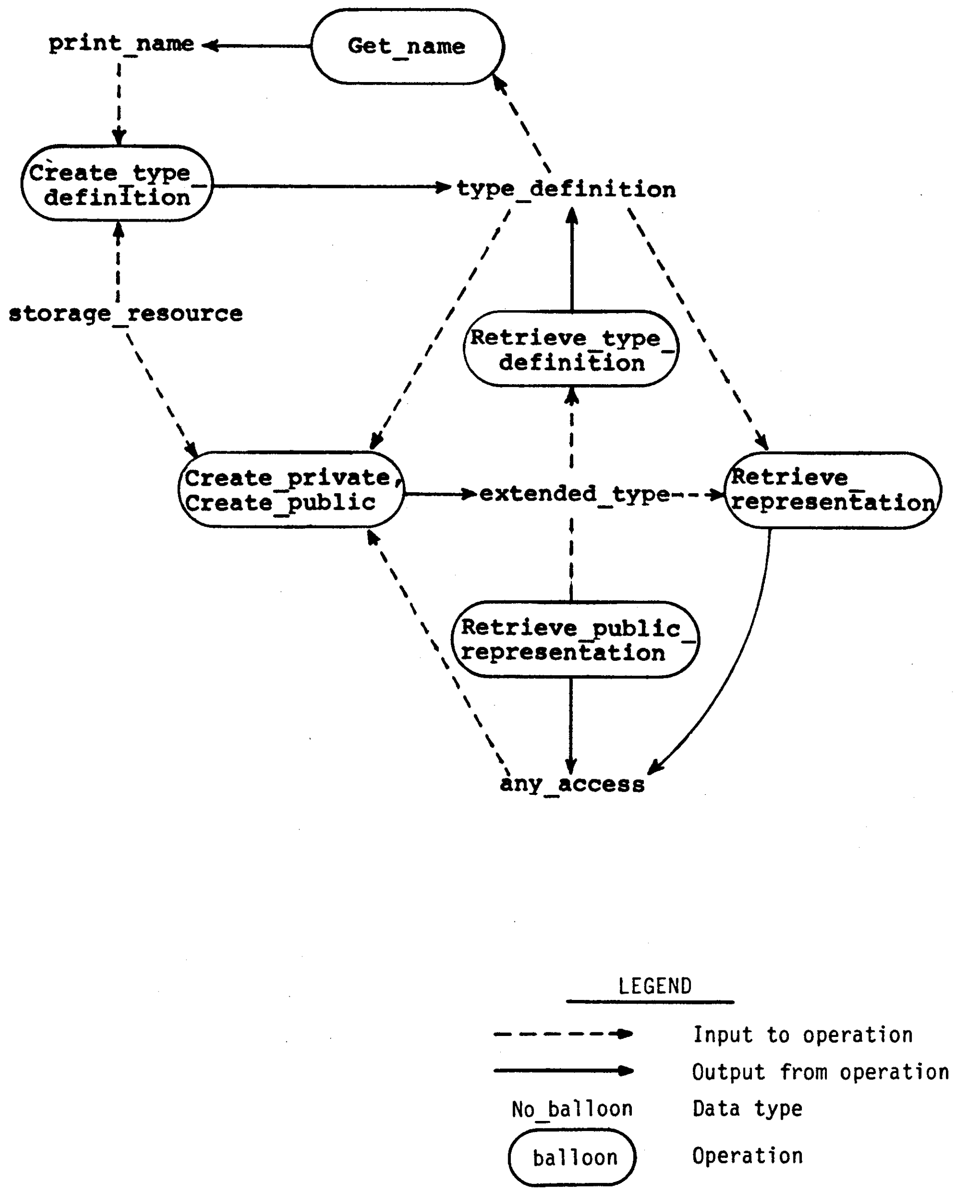
The Retrieve type definition function takes an access to an extended-type object and returns an access for its type definition. The returned access has no access rights.

The Retrieve public representation function takes an access to a public extended-type object and returns an access for its type representation. The returned access has whatever rights were specified for the type representation when the extended-type object was created. If passed an access for a non-public extended-type object, the function raises constraint_error.

The Retrieve representation function takes an access to an extended-type object and an access for a type definition that must match the type definition of the extended-type object. The function returns an access for the type representation of an extended-type object. The returned access has whatever rights were specified for the type representation when the extended-type object was created. If the type definitions don't match, then constraint_error is raised. If the extended-type object is private and the type definition access presented does not have retrieve rights, then constraint_error is raised.

If passed an access that is null or not for an extended-type object, all of these operations raise constraint_error.

Figure EXT-2 illustrates the operations and data types of the Extended Type_Manager package.



F-0290

Figure EXT-2. Extended_Type_Manager operations

## EXTENDED-TYPE OBJECT RIGHTS

The access rights on an access to an extended-type object are not used by the extended-type operations to restrict rights to the underlying object (though they may still be interpreted by user software). This is because the Retrieve_public_representation and Retrieve_representation operations return accesses for the underlying objects with whatever rights were specified when the extended-type objects were created, independent of the rights present or absent in the access to the extended-type object. For example, in Figure EXT-3, context C has an access giving it only read rights to a public extended-type object. But when C executes Retrieve_public_type_representation, an access is returned with read and write rights to the underlying object.

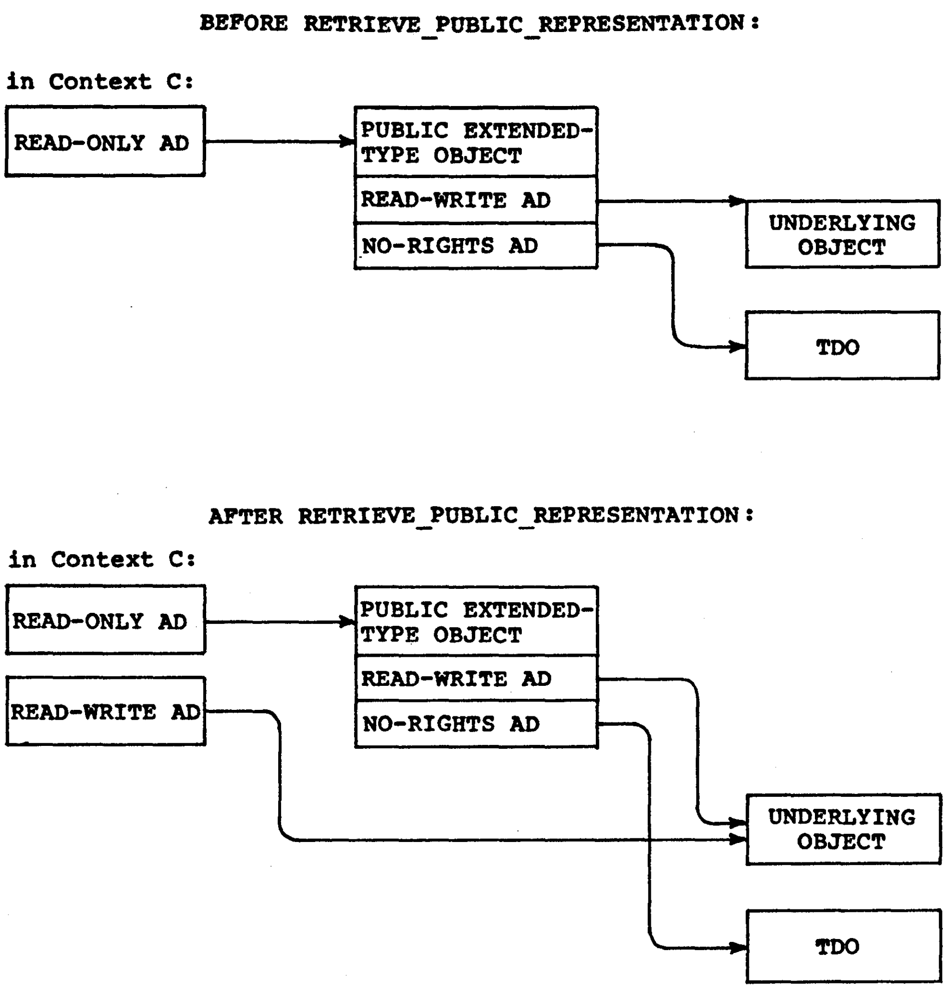


Figure EXT-3. Extended-Type Object Rights Example F-0377

## EXTENDED TYPE MANAGER PACKAGE

```
with Descriptor_Definitions, iMAX_Definitions;
package Extended_Type_Manager is

  -- Function:
  --   This package provides operations for creation and manipulation of
  --   type_definition objects and extended_type objects.
  --   The only attribute that can be associated with a type_definition
  --   in version 2 is a printable name for the type.


  use iMAX_Definitions;


  create_rights:    constant Descriptor_Definitions.rights :=
                              Descriptor_Definitions.type_right_1;
  retrieve_rights:  constant Descriptor_Definitions.rights :=
                              Descriptor_Definitions.type_right_2;


  function Create_type_definition(
      n:    print_name;            -- only attribute of type definition
      sro:  storage_resource := null)  -- SRO used in creation
    return type_definition;        -- created type definition object
                                   -- with all access rights on

    -- Function:
    --   A type_definition object is created with the specified
    --   print_name as its only attribute and with all access rights on.
    --   The SRO to be used in the creation can be specified, but
    --   defaults to the default_global_heap_sro of the calling process.


  function Get_name(
      type_def:  type_definition)  -- type_definition object whose
                                   -- print_name is requested
    return print_name;             -- attribute of the type definition

    -- Function:
    --   The print_name attribute of the specified type_definition
    --   is returned to the caller.
```

```
function Retrieve_type_definition(
   ext_type:  extended_type)  -- extended_type object
  return type_definition;     -- type_definition without create and
                              -- retrieve rights

  -- Function:
  --   This function returns the type_definition object of the
  --   specified extended_type object.  The returned type_definition
  --   object access descriptor has create_rights and retrieve_rights
  --   removed.


function Create_private(
   type_rep:  any_access;       -- representation of the extended_type
                                -- object to be created
   type_def:  type_definition;  -- type_definition object with
                                -- create_rights
   sro:       storage_resource := null)  -- SRO used in creation
  return extended_type;  -- the created private extended_type object

  -- Function:
  --   A private extended_type object is created.  Its
  --   type_representation and type_definition objects are the ones
  --   specified in the operation.  The specified type_definition
  --   object access descriptor must have create_rights.  If not, a
  --   constraint_error exception is raised.  The SRO to be used in
  --   the creation can be specified but defaults to the
  --   default_global_heap_sro of the calling process.

  -- Exceptions:
  --   constraint_error


function Create_public(
   type_rep:  any_access;  -- representation of the extended_type
                           -- object to be created
   type_def:  type_definition;  -- type_definition object with
                                -- create_rights
   sro:       storage_resource := null)  -- SRO used in creation
  return extended_type;     -- the created public extended_type object

  -- Function:
  --   A public extended_type object is created.  It has as
  --   representation and as type_definition object the ones
  --   specified in the operation.  The type_definition access
  --   descriptor that is specified must have create_rights.  If not,
  --   a constraint_error exception is raised.  The SRO to be used in
  --   the creation can be specified but defaults to the
  --   default_global_heap_sro of the calling process.

  -- Exceptions:
  --   constaint_error
  function Retrieve_public_representation(
      ext_type:   extended_type)    -- public extended_type object whose
                                    -- representation is to be retrieved
    return any_access;              -- representation of the specified
                                    -- extended_type object

    -- Function:
    --   This operation specifies a public extended_type object and
    --   returns its representation.


  function Retrieve_representation(
      ext_type:  extended_type;    -- extended_type object whose
                                   -- representation is to be retrieved
      type_def:  type_definition)  -- type_definition used in retrieval
    return any_access;             -- representation of the specified
                                   -- extended_type object

    -- Function:
    --   In case the specified extended_type object is public, its
    --   representation is returned if the specified type_definition
    --   object is the same as the one in the extended_type object.
    --   When different, a constraint_error exception is raised.
    --   In case the specified extended_type object is private, the
    --   specified type_definition object access descriptor must have
    --   retrieve_rights.  If retrieve_rights are not present, a
    --   constraint_error exception is raised.  The representation of
    --   the specified extended_type object is returned if the
    --   specified type_definition object is the same as in the
    --   extended_type object.  When different, a constraint_error
    --   exception is raised.

    -- Exceptions:
    --   constraint_error

end Extended_Type_Manager;
```

STACK MANAGER PACKAGE

```
with Descriptor_Definitions, iMAX_Definitions;
package Stack_Manager is

  -- Function:
  --   Type manager for stacks of items.  Items are handled via accesses
  --   to them.  This is analogous to the 432 port mechanism's handling
  --   of messages via accesses to the messages.  Null items are
  --   allowed.

  --   Two levels of operations are defined.  The first level defines
  --   the representation of stacks and the operations on those
  --   representations.  The user can choose to use this first level to
  --   create and operate on stack representations directly.  These
  --   direct operations are fast but unsafe.  The direct operations
  --   are unsafe  because: (1) The user of the direct operations has
  --   an access for the stack representation with read and write
  --   rights, and can access the internals of the representation
  --   (e.g., internal counts or indices) outside of this package,
  --   possibly corrupting the stack's structure.  (2) There is no
  --   run-time difference between a stack representation and some other
  --   group of segments.  For example, a user could inadvertently
  --   transfer data from an I/O device into a stack data segment.  (3)
  --   There is no way to restrict the right to perform stack
  --   operations, e.g., to give a module an access for a stack that
  --   allows the module to push items but not to pop items or to reset
  --   the stack.

  --   The second level of stack operations makes stacks a distinct
  --   extended type, with operations on stacks performed only within
  --   the Stacks package.  No using module can access the
  --   representation of stacks at this second level.  The second level
  --   also defines stack-specific access rights that are required to
  --   perform certain stack operations.  This level of stack
  --   operations is slowed by the use of extended types, but provides
  --   the same 100% protection that iMAX and the architecture provide
  --   for 432 system objects, such as ports and processes.

  --   The basic operations provided by both levels are the same, and
  --   second level operations are implemented by calling first-level
  --   operations.  Thus the substantive code in the second level is
  --   entirely devoted to handling the extended typing and rights
  --   provided by the second level.

  --   Synchronization of concurrent stack operations is not provided by
  --   this implementation and is the responsibility of package
  --   users.

  --   Summary of operations (both levels):
  --     Create  Length  Free  In_use  Push  Pop  Reset
--   Note that no operation is provided to create a stack from the
--   process stack SRO.


use Descriptor_Definitions, iMAX_Definitions;


overflow:   exception;  -- raised on push to full stack
underflow:  exception;  -- raised on pop from empty stack


package Stack_Reps is

  -- Function:
  --   Define the representation of stacks and the direct operations
  --   on stack representations.


  type stack_rep_array_index is
    new short_ordinal range 0 .. 16380;
    -- index type for stack_rep_array

  type stack_rep_array is array(stack_rep_array_index) of any_access;
    -- An array of this type holds the accesses in a stack.  Though
    -- this is a fixed array, it may be mapped onto a smaller access
    -- segment.  The bound field of stack_rep_ds contains the actual
    -- upper bound for the array.

  type stack_rep_ds is         -- stack_rep data segment
    record
      bound:     short_ordinal;-- index of last available entry in
                               -- space array (defined below)
      sp:        short_ordinal;-- index of next free entry in space
                               -- array
                               -- (<= (bound + 1))
      name:      print_name;   -- printable name of stack_rep
    end record;

  type stack_rep_ds_ref is access stack_rep_ds;
  -- needed for following record definition

  type stack_rep_as is         -- stack_rep_access segment
    record
      ds:      stack_rep_ds_ref; -- reference to data segment
      space:   stack_rep_array;  -- space for stack of items
    end record;

  type stack_rep is access stack_rep_as;
```

```
function Create(
    item_count:  short_ordinal range 0 .. 16381;
    name:        print_name := (others => ' ');
    sro:         storage_resource := null)
  return stack_rep;

  -- Function:
  --    Create a stack with space for the specified number of items
  --    with the specified name and from the specified SRO.  If the
  --    SRO is defaulted to null, the default global heap SRO
  --    referenced by the calling process's globals access segment
  --    is used.  The new stack is initially empty.  The returned
  --    access has all access rights.


function Length(stk:  stack_rep)
  return short_ordinal;

  -- Function:
  --    Return the maximum number of items that the stack can hold.

  -- Exceptions:
  --    constraint_error  -- raised if stk is null


function Free(stk:  stack_rep)
  return short_ordinal;

  -- Function:
  --    Return the number of free entries in the stack (i.e., how
  --    many more items can be pushed onto the stack without
  --    overflow?).

  -- Exceptions:
  --    constraint_error  -- raised if stk is null


function In_use(stk:  stack_rep)
  return short_ordinal;

  -- Function:
  --    Return the current depth of the stack (i.e., the number of
  --    items currently on the stack, or how many items can be popped
  --    from the stack without underflow?).

  -- Exceptions:
  --    constraint_error  -- raised if stk is null
  procedure Push(
      item:          any_access;
      stk:           stack_rep);

    -- Function:
    --   Push the item onto the stack.

    -- Exceptions:
    --   constraint_error   -- raised if stk is null
    --   overflow           -- raised if the stack is full and the item
    --                      -- cannot be pushed


  procedure Pop(
      item:    out  any_access;
      stk:          stack_rep);

    -- Function:
    --   Pop an access value from the stack and assign it to item.

    -- Exceptions:
    --   constraint_error   --  raised if stk is null
    --   underflow          --  raised if the stack is empty and no
    --                      --  value can be popped


  procedure Reset(stk:    stack_rep);

    -- Function:
    --   Reset the stack to its initial empty state.  Any items on the
    --   stack when reset is called are lost.

    -- Exceptions:
    --   constraint_error   -- raised if stk is null


private
  pragma inline(Create, Length, Free, In_use, Push, Pop, Reset);

end Stack_Reps;
```

```
package Stacks is

  -- Function:
  --   Define operations on the extended type stack.

  type stack is new iMAX_Definitions.extended_type;

  -- type rights for stacks
  push_rights: constant rights := type_right_1;
  pop_rights:  constant rights := type_right_2;
  reset_rights:constant rights := type_right_3;
  -- rights required to push, pop, or reset respectively.

  -- corresponding exceptions
  no_push_rights:  exception;
  no_pop_rights:   exception;
  no_reset_rights: exception;

  function Create(
      item_count:  short_ordinal range 0 .. 16381;
      name:        print_name := (others => ' ');
      sro:         storage_resource := null)
    return stack;

    -- Function:
    --   Create a stack with space for the specified number of items
    --   with the specified name and from the specified SRO.  If the
    --   SRO is defaulted to null, the default global heap SRO
    --   referenced by the calling process's globals access segment
    --   is used.  The new stack is initially empty.  The returned
    --   stack access has all access rights.


  function Length(stk:  stack)
    return short_ordinal;

    -- Function:
    --   Return the maximum number of items that the stack can hold.

    -- Exceptions:
    --   constraint_error  -- raised if stk is null
```

```
function Free(stk:  stack)
  return short_ordinal;

  -- Function:
  --   Return the number of free entries in the stack (i.e., how
  --   many more items can be pushed onto the stack without
  --   overflow?).

  -- Exceptions:
  --   constraint_error  -- raised if stk is null


function In_use(stk:  stack)
  return short_ordinal;

  -- Function:
  --   Return the current depth of the stack (i.e., the number of
  --   items currently on the stack, or how many items can be popped
  --   from the stack without underflow?).

  -- Exceptions:
  --   constraint_error  -- raised if stk is null


procedure Push(
    item:         any_access;
    stk:          stack);

  -- Function:
  --   Push the item onto the stack.

  -- Exceptions:
  --   constraint_error  -- raised if stk is null
  --   no_push_rights    -- raised if access stk lacks push rights.
  --   overflow          -- raised if the stack is full and the item
  --                     -- cannot be pushed
```

```
    procedure Pop(
       item:    out  any_access;
       stk:          stack);

      -- Function:
      --   Pop an access value from the stack and assign it to item.

      -- Exceptions:
      --   constraint_error  --  raised if stk is null
      --   no_pop_rights    -- raised if the access stk lacks pop rights.
      --   underflow        --  raised if the stack is empty and no value
      --                    --  can be popped


    procedure Reset(stk:    stack);

      -- Function:
      --   Reset the stack to its initial empty state.  Any items on the
      --   stack when reset is called are lost.

      -- Exceptions:
      --   constraint_error  -- raised if stk is null
      --   no_reset_rights   -- raised if access stk lacks reset rights.


  private
    pragma inline(Create, Length, Free, In_use, Push, Pop, Reset);

  end Stacks;

end Stack_Manager;
```

## STACK MANAGER PACKAGE BODY

```
with Descriptor_Definitions, iMAX_Definitions,
     Unchecked_conversion, SRO_Manager, Extended_Type_Manager;
package body Stack_Manager is


  use Descriptor_Definitions, iMAX_Definitions;


  package body Stack_Reps is


    function Create(
        item_count:  short_ordinal range 0 .. 16381;
        name:        print_name := (others => ' ');
        sro:         storage_resource := null)
      return stack_rep
    is

      s:  stack_rep;

    begin
      SRO_Manager.
        Create_access_segment(4*(item_count+1) - 1,
                              access_sgt(s), sro);
      SRO_Manager.
        Create_data_segment(short_ordinal(stack_rep_ds'size)/8 - 1,
                            data_sgt(s.ds), sro);
      s.ds.bound := item_count - 1;
      s.ds.sp := 0;
      s.ds.name  := name;
      return s;
    end Create;


    function Length(stk:  stack_rep)
      return short_ordinal
    is

    begin
      return stk.ds.bound + 1;
    end Length;


    function Free(stk:  stack_rep)
      return short_ordinal
    is

    begin
      return stk.ds.bound - stk.ds.sp + 1;
    end Free;
    function In_use(stk:  stack_rep)
      return short_ordinal
    is

    begin
      return stk.ds.sp;
    end In_use;


    procedure Push(
        item:         any_access;
        stk:          stack_rep)
    is

    begin
      if stk.ds.sp > stk.ds.bound then
        RAISE overflow;

      else
        stk.space(stack_rep_array_index(stk.ds.sp)) := item;
        stk.ds.sp := stk.ds.sp + 1;
      end if;
    end Push;


    procedure Pop(
        item:    out  any_access;
        stk:          stack_rep)
    is

    begin
      if stk.ds.sp = 0 then
        RAISE underflow;

      else
        stk.ds.sp := stk.ds.sp - 1;
        item := stk.space(stack_rep_array_index(stk.ds.sp));
      end if;
    end Pop;


    procedure Reset(stk:    stack_rep) is

    begin
      stk.ds.sp := 0;
    end Reset;

  end Stack_Reps;
```

```
package body Stacks is

  use Stack_Reps, Extended_Type_Manager;

  function Retype_to_stack_rep
  -- (a: any_access)
  -- return stack_rep;
  is new Unchecked_Conversion(any_access, stack_rep);


  stack_type:  type_definition := null;
  -- The Create function below checks to see if the TDO hasn't been
  -- created yet, and creates it when first called.


  function Create(
      item_count:  short_ordinal range 0 .. 16381;
      name:        print_name := (others => ' ');
      sro:         storage_resource := null)
    return stack
  is

    rep:  stack_rep;

  begin
    if stack_type = null then
      stack_type := Create_type_definition("stack          ");
    end if;

    rep := Stack_Reps.Create(item_count, name, sro);
    return stack(Create_private(any_access(rep), stack_type, sro));
  end Create;


  function Length(stk:  stack)
    return short_ordinal
  is

    rep:  stack_rep;

  begin
    rep := Retype_to_stack_rep(
      Retrieve_representation(any_access(stk), stack_type));
    return Stack_Reps.Length(rep);
  end Length;
```

```
function Free(stk:  stack)
  return short_ordinal
is

  rep:  stack_rep;

begin
  rep := Retype_to_stack_rep(
    Retrieve_representation(any_access(stk), stack_type));
  return Stack_Reps.Free(rep);
end Free;


function In_use(stk:  stack)
  return short_ordinal
is

  rep:  stack_rep;

begin
  rep := Retype_to_stack_rep(
    Retrieve_representation(any_access(stk), stack_type));
  return Stack_Reps.In_use(rep);
end In_use;


procedure Push(
    item:         any_access;
    stk:          stack)
is

  rep:  stack_rep;

begin
  if not Permits(any_access(stk), push_rights) then
    RAISE no_push_rights;

  else
    rep := Retype_to_stack_rep(
      Retrieve_representation(any_access(stk), stack_type));
    Stack_Reps.Push(item, rep);
  end if;
end Push;
```

```
    procedure Pop(
        item:    out  any_access;
        stk:          stack)
    is

      rep:  stack_rep;

    begin
      if not Permits(any_access(stk), pop_rights) then
        RAISE no_pop_rights;

      else
        rep := Retype_to_stack_rep(
          Retrieve_representation(any_access(stk), stack_type));
        Stack_Reps.Pop(item, rep);
      end if;
    end Pop;


    procedure Reset(stk:    stack) is

      rep:  stack_rep;

    begin
      if not Permits(any_access(stk), reset_rights) then
        RAISE no_reset_rights;

      else
        rep := Retype_to_stack_rep(
          Retrieve_representation(any_access(stk), stack_type));
        Stack_Reps.Reset(rep);
      end if;
    end Reset;

  end Stacks;
end Stack_Manager;
```

# CHAPTER IO INPUT/OUTPUT

This chapter describes iMAX packages for data transfers between peripheral devices and the 432 central system.

The synchronous interface facilities described in this chapter can be used to implement higher-level I/O facilities such as the Ada TEXTIO package.

The asynchronous interface facilities described in this chapter are needed only by users implementing device drivers, or by users with special I/O requirements. Other readers of this chapter may want to skip the Asynchronous Interface portion.

Figure IO-1 illustrates the flow of input/output transactions between a user process in a 432 central system and a device monitor task that manages a physical device in a Peripheral Subsystem (PS). A 432 Interface Processor (IP) connects the two systems and is managed by iMAX IP controller software running in the Peripheral Subsystem. All of the software running in the Peripheral Subsystem, as well as how to interface additional devices to iMAX, are described in the next chapter, IOI, Input/Output Implementation. This chapter describes I/O from the viewpoint of a user writing programs to run on the 432 central system.

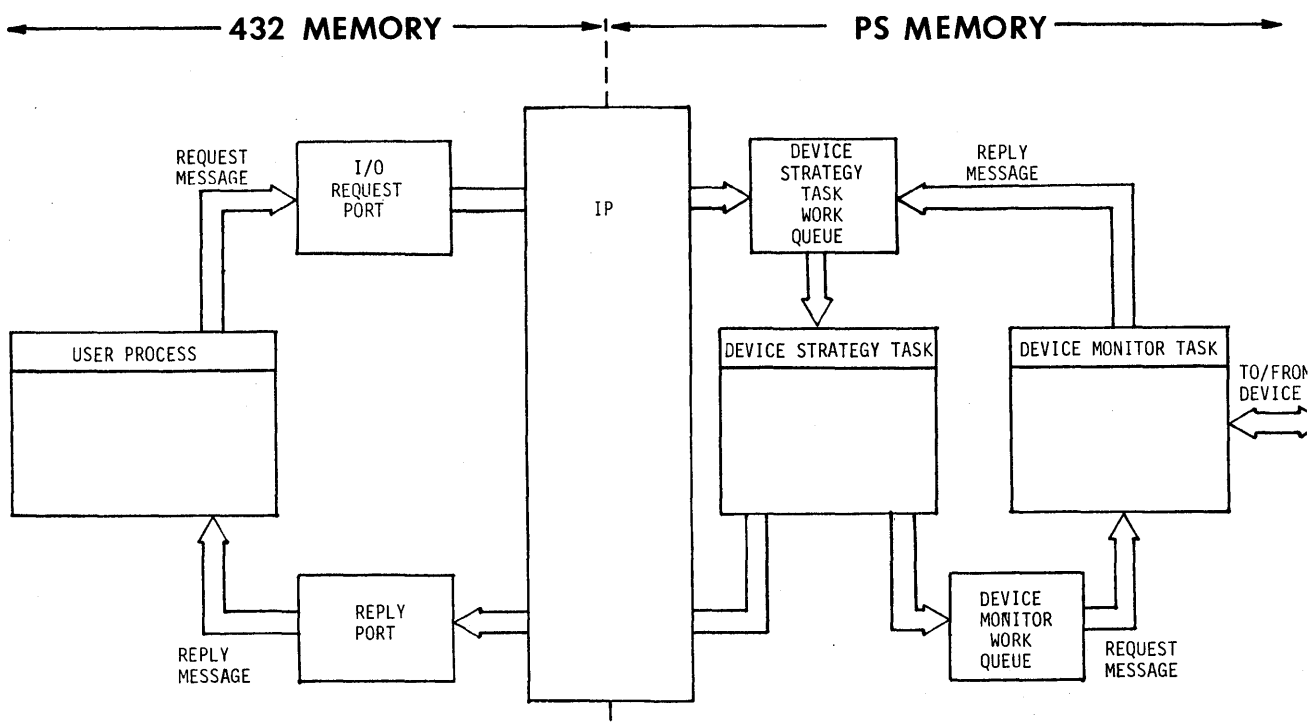


Figure IO-1. Input/Output Flow

iMAX I/O is based on an extensible I/O philosophy that will be followed in all subsequent releases of iMAX. This I/O model supports both new devices and new types of devices (new device abstractions). One device interface can support more than one abstraction. For example, a graphics terminal could support both a "plain terminal" abstraction and a "graphics display" abstraction. An abstraction is a collection of operations, supported by one or more device types, that forms a useful abstract device. For example, a simple terminal provides "Read" and "Write" operations, while the graphics display abstraction might add such operations as "Draw_line". The "device" that underlies the operations of an abstraction may actually be defined only by software and not correspond to any physical device or file.

The iMAX I/O interfaces treat data to be transferred as sequences of bytes grouped into buffers. Higher-level interfaces, such as the Ada package TEXTIO, can be provided with each language and implemented using the iMAX interface.

iMAX does not support the creation or deletion of devices or device interfaces in a running system. All device interfaces are available at system initialization.

iMAX does not manage concurrent access to shared devices. For example, if two processes both write reports to a shared printer, then use of the printer needs to be coordinated. Otherwise, one process could write part of its output in the middle of a report from the other process.

## SYNCHRONOUS INTERFACE

These packages provide interfaces to iMAX synchronous I/O:

| | |
|---|---|
| • IO_Definitions | -- type declarations used by the<br>-- interfaces |
| • Synchronous_IO_Interfaces | -- a standard procedural interface<br>-- to devices |
| • Terminal_Interfaces | -- an extension of the standard<br>-- procedural interface to support<br>-- terminals |
| • Debug_Source and Debug_Sink | -- instances of the standard<br>-- procedural interface that<br>-- use the DEBUG-432<br>-- debugger console for I/O |
| • IO_Devices | -- references for device<br>-- interfaces provided by iMAX |

This section describes these packages and concludes with a simple example of using the synchronous I/O interface.

## DEFINITIONS

The package IO Definitions gives type declarations used by the various I/O interfaces.

The access type query record consists of accesses for records that describe device characteristics, of type query record rep. These device characteristics include:

- a printable name for the device,
- a preferred buffer length (represented as length minus one)
- whether buffers can vary in length or must be a multiple of the preferred buffer length
- a unique device number
- an identification number for the Attached Processor through which requests are sent to the device
- an array of "abstractions" supported by the device, where the abstractions are simply the different procedural interfaces supported by the device.

## GENERIC SYNCHRONOUS INTERFACES

The package Synchronous IO Interfaces defines the standard procedural interfaces to generic devices in the iMAX I/O model. Three package types are defined:

- Source -- operations required for all source devices (input-only)
- Sink -- operations required for all sink devices (output-only)
- Store -- operations required for all store devices

More than one of these package types can contain the same operation with the same meaning. For example, the Read operation is defined by the two package types Source and Store. Table IO-1 shows which operations are contained in which of the package types, and is followed by descriptions of the operations. Each operation is described only once, even if it occurs in more than one package type.

Those operations that are required for all devices (and thus common to Source, Sink, and Store) are collected in the dummy package type Basic_IO_Interface. This package type is defined only by comments and serves as a centralized description of the common operations.

Table IO-1. Synchronous_IO_Interfaces Package Types and Operations

| | Package Types | | |
|---|---|---|---|
| Operations | Source | Sink | Store |
| Interface_description | X | X | X |
| Close | X | X | X |
| Reset | X | X | X |
| Transform_interface | X | X | X |
| Get_asynchronous interface | X | X | X |
| Flush | | X | X |
| Read | X | | X |
| Write | | X | X |

X = operation is present in package type

The Interface description function has no parameters. Interface description returns an access of type query record that references a description of the device. Query records are defined by the IO Definitions package which is described above in the section I/O Definitions. The query record includes identification information, buffering information, and an array of supported abstractions for the device.

The Close procedure has no parameters. Close indicates that the user does not want to use the interface anymore. Close does not affect requests made before a Close. Close, Flush, Read, or Write requests made after a Close raise the operation_not_allowed exception. Close is the only way to signal an end-of-file condition to an output device. To use the device to output a subsequent file, the device must be reset.

The Reset procedure has no parameters. Reset aborts any ongoing or queued operations using the device and places the device into a consistent state. The exact effect of Reset is device-dependent.

**NOTE**

For all physical devices, including all the device interfaces provided by iMAX V2, Close performs no operation. In the device interfaces provided by iMAX V2, Reset also performs no operation.

The Transform interface function takes one parameter, new interface type, of type sink transform types or source transform types defined by the IO_Definitions package. Different values of these transform_types type correspond to different package types. For example, the value transform_to_source corresponds to the Source package type. If the designated package type is supported by the device, an any access value is returned which references a package of that type for the specific device. The caller must use Unchecked_Conversion to retype the returned access to the proper type. iMAX has already defined these instances of Unchecked__Conversion to support the user: Any_access_to_Source, Any_access_to_Sink, Any_access_to_Terminal_Source, and Any_access_to_Terminal_Sink. These instances are defined in the Terminal_Interfaces package, described below in the section Synchronous Terminal Interfaces. If Transform_interface is called with a request for a package type not supported by the device, then the transformation_not_allowed exception is raised. **Note: Until Ada exceptions are implemented, Transform_interface returns a null if an invalid transformation is requested (instead of raising the transformation_not_allowed exception).

The Get asynchronous interface function has no parameters. It returns an access of type Asynchronous_IO_Interface.connection for the connection object that represents the asynchronous interface to the device. The access returned has read rights and all type rights, but no write rights. Connection objects and the asynchronous I/O interface are described below in the section Asynchronous Interface. If the device does not provide an asynchronous interface (for example, the debugger I/O interface doesn't), then Get__asynchronous__interface returns null.

The Flush procedure, defined for Sinks and Stores, has no parameters. Flush does not return to its caller until all previously-written data has actually been written by the device (the Write procedure can return after simply queuing the data to be written).

An example of when Flush would be used is a machine-tool control program that sends a warning to an operator before activating a punch press. It is important that the warning actually reach the operator before the press comes down! The program would first write the warning, then call Flush to ensure that the warning is displayed before Flush returns, then probably invoke a time delay of a few seconds, and then activate the press.

Another use of Flush is to ensure that output is written before invoking Reset for a device.

The Read procedure, defined for Sources and Stores, has four parameters. Data access is a data segment access with write rights (an access for the segment containing the input buffer). Offset is the offset in bytes from the segment base to the beginning of the buffer. Requested length is the number of bytes requested. Returned length is an output parameter that is assigned the number of bytes actually transferred. Read returns when the next block from the device has been read into the buffer.

The number of bytes returned by Read is affected by the device interface characteristics buffer_length and fixed_length. When fixed length is true, the requested length must be at least as large as the preferred buffer length, or the operation_not_allowed exception is raised. Also when fixed_length is true, the number of bytes returned will be an integer multiple of the preferred buffer length. Note that none of the device interfaces currently supplied with iMAX uses fixed length buffers. Regardless of fixed_length true or false, the number of bytes returned will be less than or equal to the number of bytes requested.

When there is no more data to be read, returned_length is set to zero and the end_of_file exception is raised.

If a hardware or software error causes the transfer to fail, the buffer may contain no data, partial data, or erroneous data, and Read raises the transfer_error exception.

The Write procedure, defined for Sinks and Stores, has three parameters. Data access is a data segment access with read rights (an access for the segment containing the output buffer). Offset is the offset in bytes from the segment base to the beginning of the buffer. Length is the actual number of bytes to be written.

If the interface characteristic fixed_length is true, then the number of bytes to be written must be an integer multiple of the preferred buffered length, or the operation_not_allowed exception is raised.

Write is not used to signal an end-of-file condition to a device; an end-of-file can be indicated using the Close operation.

If a hardware or software error causes the transfer to fail, no data, partial data, or erroneous data may be written, and Write raises the transfer_error exception.

**NOTE**

None of the device interfaces (e.g., terminal interfaces) supplied by iMAX V2 supports the concept of an end-of-file condition. Therefore, the end_of_file exception is never raised, and a returned_length of zero occurs only if requested_length is zero.

These exceptions are defined by the Synchronous_IO_Interfaces package:

| Exception | Description |
|---|---|
| • end_of_file | No more input data from Read. In iMAX V2, this exception is not raised. |
| • transformation_not_allowed | Transform_interface was called with a transform__type corresponding to a package type not supported by the device. In iMAX V2, this exception is not raised. |
| • operation_not_allowed | The requested__length parameter to Read or the length parameter to Write is not consistent with a fixed buffer size required by the device. In iMAX V2, this exception is not raised. |
| • transfer_error | Read or Write encountered an unrecoverable hardware or software error in trying to transfer data. In iMAX V2 this exception is not raised. |

**NOTE**

In iMAX V2, conditions that would raise the operation_not_allowed or transfer_error exceptions instead cause a system error, as described in Appendix FLT, Fault-Handling.

## TERMINAL INTERFACES

The package Terminal Interfaces defines the procedural interfaces to terminals in iMAX. Two package types are defined:

- Terminal_Source -- operations available in the terminal input interface.
- Terminal_Sink -- operations available in the terminal output interface.

These package types extend the iMAX Source and Sink package types with two new operations: Get terminal characteristics and Set terminal characteristics.

Terminal Interfaces defines two types, terminal source characteristics and terminal sink characteristics which are characteristics for the input and output sides of a terminal respectively. Each is a record type with the two fields status and speed. Status has the record type terminal status with a single field terminal connected of type boolean, true if the terminal is connected. In iMAX V2, the terminal_connected field is true if and only if the corresponding terminal support software is in the system configuration. In iMAX V2, the terminal connected field cannot be changed by users. Speed has the enumeration type terminal_speed, with possible values baud150, baud300, baud600, baud1200, baud2400, baud4800, baud9600, and baud19200.

The Get_terminal_characteristics function returns a copy of the characteristics record for the source or sink side of a terminal.

The Set_terminal_characteristics procedure assigns a new value to the characteristics record for the source or sink side of a terminal. If only some characteristics are to be changed, the safest programming practice is to use Get_terminal_characteristics to copy the characteristics, change the desired entries in the copy, and then call Set_terminal_characteristics with the revised copy. Note that in iMAX V2, the Set_terminal_characteristics procedure does not change the terminal_connected field, but simply ignores the value supplied by the user.

Figure IO-2, Terminal Characteristics Examples, gives examples of reading and modifying terminal characteristics.

The Close and Reset procedures defined for terminals perform no operation.

(1) To determine whether the input side of terminal 0 is connected:

```
if  IO_Devices.terminals(0).tsrc.
      Get_terminal_characteristics().status.
      terminal_connected
  then
    .
    .
    .
```

(2) To change the baud rate of the output side of terminal 3 without changing any other characteristics:

```
declare
  t:  Terminal_Interfaces.terminal_sink_characteristics;
begin
  t := IO_Devices.terminals(3).tsnk.
       Get_terminal_characteristics();

  t.speed := Terminal_interfaces.baud1200;

  IO_Devices.terminals(3).tsnk.Set_terminal_characteristics(t);
end;
```

Figure IO-2. Terminal Characteristics Examples

**NOTE**

In iMAX V2, the terminal interface generates a system error if any Read operation does not succeed. System errors are described in Appendix FLT, Fault-Handling. Also, the Write operation may not check to see whether the write succeeds but assume that it does. This information may be needed by users modifying the iMAX terminal support software.

The interface_description function for terminal sources and sinks returns the query_record information shown in Table IO-2. The value returned for the field abstractions_array depends on whether a Source or Sink package is being used.

Table IO-2. Terminals query_record Information

| Field | Type | Value | Comments |
|---|---|---|---|
| device_name | print_name | "term xxx " | -- xxx is decimal -- terminal number |
| buffer_length | short_ordinal | 135 | -- 136 bytes maximum |
| fixed_length | boolean | false | -- variable-length -- buffers |
| device_number | short_ordinal | * | -- unique device ID |
| AP_number | short_ordinal | * | -- unique AP ID |
| abstractions | abstractions_ array | (transform_to_source, transform_to_terminal_source) or (transform_to_sink, transform_to_terminal_sink) | |

## Terminal Handler

iMAX V2 terminal support uses the iRMX 88 terminal handler. Table IO-3 lists control characters recognized from the keyboard by the terminal handler.

Table IO-3. Terminal Handler Control Characters

| Keyboard Character | ASCII Code | Effect |
|---|---|---|
| rubout | DEL (16#7F#) | deletes the previous character (if any) of the current line |
| control-X | CAN (16#18#) | deletes all characters on the current line |
| return | CR (16#0D#) | echoes carriage return/ linefeed and appends a linefeed (but no carriage return) to the input line |
| control-S | DC3 (16#13#) | suspends terminal output |
| control-Q | DC1 (16#11) | resumes terminal output |

The only character given special interpretation on output is ASCII linefeed (16#0A#), which is translated to a carriage return/linefeed sequence (carriage return = 16#0D#). Linefeed is not recognized as a logical break character on input.

iMAX defines a logical line as a variable-length sequence of ASCII characters terminated by a linefeed. The linefeed character is considered to be part of the line that it terminates. The logical line is expected to form the basis for a standard iMAX text file format.

A terminal input buffer is filled with either the requested number of characters or with the next logical line, whichever is shorter. If a logical line is returned, it is terminated with a linefeed.

### Conversion Routines

The Terminal_Interfaces package defines four conversion functions that are used with the Transform_interface operation: Any_access_to_Source, Any_access_to_Terminal_Source, Any_access_to_Sink, and Any_access_to Terminal_Sink. Each takes one any_access parameter and converts it to the package type named.

## DEBUGGER I/O INTERFACE

This section describes the iMAX I/O interface to the DEBUG-432 debugger's console device. This debugger I/O interface is the only I/O interface available in the minimal configuration of iMAX. The debugger I/O interface provides a Debug Source package, of package type Source, for input, and a Debug Sink package, of package type Sink, for output.

The DEBUG-432 debugger executes on the Intellec Series III Microcomputer Development System within the Intel 432 Cross Development System. DEBUG-432 is described in the Intel 432 Cross Development System Workstation User's Guide.

Debugger I/O uses one IP in a 432/670 Execution Vehicle that is dedicated to use by DEBUG-432 and that operates in physical mode. The iMAX interface to debugger I/O is shown in Figure IO-3.

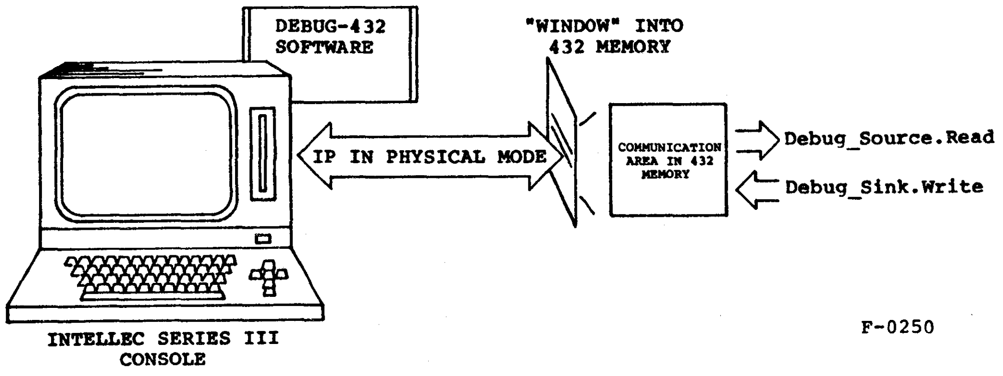


Figure IO-3. Debugger I/O Interface

The Interface_description function for Debug_Source and Debug_Sink returns the query_record information shown in Table IO-4. The value returned for the abstractions_array field depends on whether a Source or Sink package is being used.

Table IO-4. Debugger I/O query_record Information

| Field | Type | Value | Comments |
|---|---|---|---|
| device_name | print_name | "debuggerIO " | -- 132 bytes<br>-- maximum |
| buffer_length | short_ordinal | 131 | -- 132 bytes<br>-- maximum |
| fixed_length | boolean | false | -- variable-length<br>-- buffers |
| device_number | short_ordinal | 1 | -- unique device ID |
| AP_number | short_ordinal | 1 | -- unique AP ID |
| abstractions | abstractions_ array | transform_to_source<br>or<br>transform_to_sink | -- value if source<br>-- used for call.<br>-- value if sink<br>-- used for call. |

The Close and Reset procedures defined by Debug_Source and Debug_Sink perform no operation.

There is no asynchronous interface to debugger I/O, and the Get_asynchronous_interface function defined by Debug_Source and Debug_Sink returns the access value null.

The debugger I/O interface is fully synchronous -- neither Read nor Write return until the operation is complete. A process doing a Read or Write sets up the operation and sets a flag to indicate to the debugger that an operation is requested. The requesting process then loops, checking another flag, until the debugger signals that the transfer is accomplished. Each time the process checks the flag unsuccessfully, it calls iMAX_Definitions.Idle(0). This Idle call reschedules the process, allowing other processes an opportunity to run. Because a process loops, "busy waiting", while waiting for a debugger I/O transfer, use of debugger I/O can impact system performance dramatically. If there is no debugger polling for I/O, then the caller will wait forever.

## Console Handler

Debugger I/O uses the ISIS-II console handler. Table IO-5 lists control characters recognized from the keyboard by the console handler.

Table IO-5. Console Handler Control Characters

| Keyboard Character | ASCII Code | Effect |
|---|---|---|
| rubout | DEL (16#7F#) | deletes the previous character (if any) of the current line |
| control-X | CAN (16#18#) | deletes all characters on the current line |
| return | CR (16#0D#) | echoes carriage return/ linefeed and appends a linefeed (but no carriage return) to the input line |
| control-S | DC3 (16#13#) | suspend terminal output |
| control-Q | DC1 (16#11#) | resume terminal output |

The only character given special interpretation on output is ASCII linefeed (16#0A#), which is translated to a carriage return/linefeed sequence (carriage return = 16#0D#). Linefeed is not recognized as a logical break character on input.

A debugger console input buffer is filled with either the requested number of characters or with the next logical line, whichever is shorter. If a logical line is returned, it is terminated with a linefeed.

## IO DEVICES

The package IO_Devices provides access to the device interfaces provided by iMAX.

iMAX supports 0, 1, or 5 terminals in its default I/O configurations. For each terminal, there is an entry in a terminals array provided by IO Devices. Each entry is an AD to a record with four components:

- source — an access for a Source package for that terminal
- sink — an access for a Sink package for that terminal
- tsrc — an access for a Terminal_Source package for that terminal
- tsnk — an access for a Terminal_Sink package for that terminal

The first terminal is in entry 1, the next in entry 2, etc. Unused entries are null. Note that entry 0 is unused and null in all three default I/O configurations.

IO_Devices defines the variables debug source and debug sink which are accesses for a Source package for the debugger console and a Sink package for the debugger console. IO_Devices.debug_source is the same as the Debug_Source package; IO_Devices.debug_sink is the same as the Debug_Sink package.

IO_Devices defines the console variable which is an access for a Sink package to which system messages are written. The user can reassign the console. The console is initially assigned to the debugger console in both iMAX configurations.

Note that iMAX V2 does not define any interfaces of type Store.

## SYNCHRONOUS INTERFACE EXAMPLE

This section gives an example of using the synchronous interface, a package Hello_World with one procedure Main that writes the message "HELLO WORLD" to the debugger console.

### Hello World Package

```
    package Hello_World is

  -- Function:
  --    Say "HELLO WORLD" to the debug console via procedure Main.

  procedure Main;

    -- Function:
    --    Say "HELLO WORLD" via the debug console interface.

end Hello_World;
```

Hello World Package Body

```
with iMAX_Definitions, Debug_Sink, Unchecked_Conversion;
package body Hello_World is

  procedure Main is

    line_length:  constant integer := 80;
    type buffer_rep is new string(1..line_length);
    type buffer is access buffer_rep;

    function Retype_buffer_to_any_ds
      -- (b: buffer)
      -- return any_ds;
      is new Unchecked_conversion(buffer, iMAX_Definitions.any_ds);

      -- Function:
      --   The given access for a buffer is retyped to any_ds.
      --   This operation does not affect the rights on the access.


    line:  buffer := new buffer_rep;

  begin
    line(1..11) := "HELLO WORLD";
    line(12) := ASCII.lf;  -- defined by Ada in Standard package;
                           -- linefeed is iMAX end-of-line character.

    Debug_Sink.Write(Retype_buffer_to_any_ds(line), 0, 12);

  end Main;

end Hello_World;
```

## ASYNCHRONOUS INTERFACE

The asynchronous interface facilities described in this section are needed only by users implementing device drivers or users with other special I/O requirements. Other readers may want to skip this section.

The asynchronous interface for a device defines a port-based connection between the device and user processes requesting I/O services from it. A process makes a request by sending a request message of the correct format to the connection. Device software receives these messages, performs requested services, and answers by sending the same message back to the connection. The message is now a reply message and indicates success or failure of the request, as well as including any data successfully requested. User software, usually the same process that sends a request, receives the reply message from a reply port.

iMAX standardizes the asynchronous interface by defining the format of both connections and the IO messages that flow through them. iMAX also defines the standard command codes and reply codes that are used in the IO_messages. These codes can be extended by users who implement their own I/O device interfaces.

For both command codes and reply codes, Intel reserves all values in the range 0..16383. Users should use values in the range 16384..32767 if they are extending the asynchronous interface.

Asynchronous_IO_Interface provides operations that parallel the 432 port operations, but that operate on connections and IO_messages. These procedures are provided: Send, Cond_send, Receive, and Cond receive. Send and Cond_send assume that the reply_port specified by the message is already valid. Receive and Cond_receive receive from the reply port specified by the connection. Programs can also directly use the port operations on the ports specified by a connection.

### CONNECTIONS

Each connection is represented by a record of type connection record which defines:

- access to a port to which request messages are sent
- a printable name for the connection
- access to the query_record that describes the device
- access to a port from which reply messages can be received

Note that the user of the connection must still store an AD for a reply port in each IO_message. The reply port provided with a connection has two purposes. First, a process can choose to use the reply port provided by the connection, avoiding the need to create a reply port of its own.

However, if more than one process uses a connection, they should either provide their own reply ports or synchronize usage of the connection. Otherwise, one process could receive a reply message intended for another process. The second use of the reply port in the connection is by the Receive and Cond_Receive operators defined for connections.

## IO_MESSAGES

An <u>IO message</u> is an access for an <u>IO message record</u> with these parts:

- access to a command record for the message
- access to a reply port to which the message will be returned
- an array of accesses for the data segments containing the buffers (if any) associated with the messages. Note: In iMAX V2, only one buffer can be associated with each IO_message.

The command record for the message contains these fields:

| | |
|---|---|
| • command | one of the command codes |
| • message_id | a unique identifier for the message, which can be used to check that the same message is received in reply, or for any other user-defined purpose. This field is preserved by the iMAX software that handles the message. |
| • reply_code | reply code defined filled in by the device monitor task on the AP |
| • buffer_descriptions | an array describing the location of the buffers within the data segments that contain them. For each buffer, offset, requested_length (actually length minus one), and_returned length (actually length minus one) are defined. The returned_length field is filled in by the device monitor task on the AP. Note: In iMAX V2, this array has only one entry and only one buffer can be associated with each IO_message. |

Table IO-6 describes the asynchronous interface command codes.

Table IO-7 describes the asynchronous interface reply codes.

Table IO-8 shows what reply codes can be received for any given command code.

Table IO-6. Asynchronous Interface Command Codes

| Value | Name | Description | Abstractions Defined For: |
|---|---|---|---|
| 0 | ---- | UNUSED: RESERVED BY INTEL | ---- |
| 1 | reset | Aborts current operations of a device and places the device into a consistent state (e.g., for a printer, place the print-head at the left margin on a new page). The exact effect of reset is device-dependent. Any requests which preceded reset but were not completed are returned with the reset returned reply code. A partially completed request may either be completed or returned uncompleted. | all |
| 2 | read | Reads data from the device into the data buffer(s) referenced by the message. An end-of-file condition on the device is indicated by an end of file reply code. Additional requests after an end-of-file condition is returned have a device-dependent action, but in general will also be returned with an end_of_file reply code. | Sources, Stores |
| 3 | write | Writes data to the device from the buffer(s) referenced by the message. An end-of-file is indicated by sending a close command. | Sinks, Stores |
| 4 | close | Indicates either that the device will no longer be used, or that an output file is being closed. | all |
| 5 | flush | Ensures that all previous output data has been physically moved to the device before a reply to the flush request is sent. | Sinks, Stores |

| | | |
|---|---|---|
| 6 set_device_ characteristics | Uses the data with the request as new device characteristics (e.g., baud rate). | Terminal_ Sources, Terminal_ Sinks |
| 7 get_device_ characteristics | Copies current device characteristics into the buffer supplied with the request. | Terminal_ Sources, Terminal_ Sinks |

Table IO-7. Asynchronous Interface Reply Codes

| Value | Name | Description |
|---|---|---|
| 0 | success | The operation was successful. The rest of the command record contains valid information and any requested data is present. |
| 1 | end_of_file | An end-of-file condition exists on the input from a Source or Store. No additional data will be provided for that request. Some devices, e.g., a terminal, may signal an end-of-file for a specific command but continue to supply data. |
| 2 | not_processed | The data buffer has not been processed. This reply code is returned for requests that use data chaining and receive an end-of-file condition for a previous buffer. [Data chaining is the transfer of multiple data buffers with a single IO_message. Data chaining is not supported in iMAX V2 and this reply code is currently not used.] |
| 3 | reset_required | The device is in an inconsistent state and cannot process any normal requests until a reset command is issued. Device-dependent status commands may still be accepted. |
| 4 | invalid_command | The requested command is not a valid one for the device. |
| 5 | bad_data_ buffer_size | The buffer specified in the request is either too large or too small for the device. |
| 6 | invalid_request | The request did not contain the expected segments, e.g., the data_buffer was omitted. |

| | | |
|---|---|---|
| 7 | hard_IO_error | The device has detected an unrecoverable error. |
| 8 | interface_closed | The interface was closed and no further requests are valid. |
| 9 | reset_returned | The request is returned as a result of a reset command. The request is not satisfied. |

Table IO-8. Asynchronous Interface Commands and Replies Cross-Reference

| Reply: \ Command: | reset | read | write | close | flush | set_device_ characteristics | get_device_ characteristics |
|---|---|---|---|---|---|---|---|
| success | X | X | X | X | X | X | X |
| end_of_file | | X | | | | | |
| not_processed | | X | | | | | |
| reset_ required | | X | X | X | X | X | X |
| invalid_ command | | X | X | | X | X | X |
| bad_data_ buffer_size | | X | X | | | | |
| invalid_ request | X | X | X | X | X | X | X |
| hard_IO_ error | | X | X | | | | |
| interface_ closed | X | X | X | X | X | X | X |
| reset_ returned | | X | X | X | X | X | X |

X = command can get reply

## PACKAGE LISTINGS

This section gives listings for packages described by this chapter:

IO_Definitions
Synchronous_IO_Interfaces
Terminal_Interfaces
Debug_Source
Debug_Sink
IO_Devices
Asynchronous_IO_Interface

## IO DEFINITIONS PACKAGE

```
with iMAX_Definitions;
package IO_Definitions is

  -- Function:
  --   This package contains definitions common to the synchronous and
  --   asynchronous I/O interfaces.


  use iMAX_Definitions;

  -- Temporary parameter types for specifying the interface to
  -- transform to.

  type transform_types is new short_ordinal;

  subtype source_transform_types is transform_types range 0 .. 10000;

  transform_to_source:          constant source_transform_types := 1;
  transform_to_terminal_source: constant source_transform_types := 2;

  subtype sink_transform_types is transform_types range 10001 .. 20000;

  transform_to_sink:            constant sink_transform_types := 10001;
  transform_to_terminal_sink:   constant sink_transform_types := 10002;

  type abstraction_name_array is array (1 .. 30) of character;

  type abstraction_description_val is
    record
      abstraction_name: abstraction_name_array;
      abstraction_type: transform_types;
    end record;
  type abstraction_description is access abstraction_description_val;
  type abstractions_array is array (1 .. 10) of abstraction_description;

  type query_record_rep is
    record
      device_name:   print_name;     -- printable device identifier
      buffer_length: short_ordinal;  -- preferred buffer length minus one
      fixed_length:  boolean;        -- true if "preferred" length is
                                     -- required
      device_number: short_ordinal;  -- unique system device identifier
      AP_number:     short_ordinal;  -- controlling attached processor ID
      abstractions:  abstractions_array;  -- description of supported
                                          -- abstractions
    end record;

  type query_record is access query_record_rep;

end IO_Definitions;
```

## SYNCHRONOUS IO INTERFACES PACKAGE

```
with IO_Definitions, iMAX_Definitions, Asynchronous_IO_Interface;
package Synchronous_IO_Interfaces is

  -- Function:
  --   This package includes definitions of the standard synchronous
  --   I/O interfaces and related types.


  use IO_Definitions;

  ----------------------------------------------------------------------
  --  Exceptions
  ----------------------------------------------------------------------
  end_of_file:                 exception;  -- no more input data
  transformation_not_allowed:  exception;  -- Transform_interface was
                                           -- called with an
                                           -- unrecognized new interface
                                           -- type
  operation_not_allowed:       exception;  -- the specified operation is
                                           -- not recognized by the
                                           -- driver or support routine
  transfer_error:              exception;  -- an I/O or protocol error
                                           -- was detected during a data
                                           -- transfer

  subtype xfer_range  is integer range 0 .. 2**16;
                                       -- up to 65,636 bytes can be
                                       -- transferred with one operation


------------------------------------------------------------------------
  -- The following set of functions is required of all synchronous I/O
  -- interfaces:
  --
  --  package type Basic_IO_Interface is
  --
  --    -- Function:
  --    --   This package type defines the minimum synchronous
  --    --   interface.  It must be provided for all devices.  It
  --    --   includes only routines for determining and changing device
  --    --   interface characteristics, and for closing the interface.
--    procedure Interface_description
--      return query_record;  -- interface description for this
--                            -- interface
--
--      -- Function:
--      --   This routine returns an access to the interface
--      --   description for this interface.
--
--    procedure Close;
--
--      -- Function:
--      --   This routine renders the interface unusable, after
--      --   first flushing any buffers and completing any outstanding
--      --   operations.  Any further operations cause an error.
--
--    procedure Reset;
--
--      -- Function:
--      --   This routine reinitializes the interface.  The effect of
--      --   Reset is device-dependent.
--
--    function Transform_interface(
--        new_interface_type:  type_description)  -- type of the new
--                                                 -- interface
--      return dynamic_typed;
--
--      -- Function:
--      --   This routine returns a new,
--      --   possibly expanded or restricted view, of this I/O
--      --   interface.  The transformations allowed can be
--      --   determined by calling Get_Interface_Characteristics.
--      --!! Get_interface_characteristics is now
--      --!! Interface_description.
--      --
--      -- Note:
--      --   Dynamic_typed and type_description are not implemented
--      --   in the current compiler, so the input parameter is a
--      --   scalar type defined in IO_Definitions, and an access
--      --   descriptor for the domain of type any_access is
--      --   returned.  This AD must be converted to the correct type
--      --   using Unchecked_Conversion.  Appropriate instances of
--      --   Unchecked_Conversion are defined in Terminal_Interfaces
--      --   for the currently available devices (i.e. terminals).
--
--    function Get_asynchronous_interface
--      return Asynchronous_IO_Interface.connection;
--      --
--      -- Function:
--      --   This function returns an access for a connection object
--      --   which implements the standard asynchronous device
--      --   interface.
--
--  end Basic_IO_Interface;
```

---

```
package type Source is

   -- Function:
   --   This package type defines the synchronous source interface.


   -- The following record and three procedures are described in the
   -- Basic_IO_Interface package.

   function Interface_description
     return query_record;

   procedure Close;

   procedure Reset;

   function Transform_interface(
       new_interface_type: source_transform_types)
     return any_access;
   --  See note in description of Transform_interface given above.
   --         new_interface_type:  type_description)
   --        return dynamic_typed;

   function Get_asynchronous_interface
     return Asynchronous_IO_Interface.connection;

   -- end of basic package

   procedure Read(
       data_access:         iMAX_Definitions.any_ds;
                              -- access for segment containing buffer
       offset:              xfer_range;
                              -- offset of buffer within segment
       requested_length:    xfer_range;
                              -- number of bytes to transfer
       returned_length: out xfer_range);
                              -- number of bytes actually transfered

     -- Function:
     --   This routine does a device-dependent read operation,
     --   controlled by the interface characteristics buffer_length
     --   and fixed_length.
     --
     --   The number of bytes returned will be less than or equal to
     --   that requested when fixed_length is false.  The end_of_file
     --   exception is raised when there is no more data
     --   (returned_length is zero.)
```

```
    --
    --   The number of bytes returned is a multiple of the preferred
    --   buffer length (buffer_length plus one) when fixed_length is
    --   true.  The operation_not_allowed exception is raised if the
    --   requested_length is not at least as large as the preferred
    --   buffer length.  The end_of_file exception is raised when
    --   there is no more data (returned_length is zero.)

end Source;



package type Sink is

  -- Function:
  --   This package type defines the synchronous sink interface.


  -- The following record and three procedures are described in the
  -- Basic_IO_Interface package.

  function Interface_description
    return query_record;

  procedure Close;

  procedure Reset;

  function Transform_interface(
      new_interface_type: sink_transform_types)
    return any_access;
  --  See note in description of Transform_interface given above.
  --        new_interface_type:  type_description)
  --      return dynamic_typed;

  function Get_asynchronous_interface
    return Asynchronous_IO_Interface.connection;

  -- end of basic package

  procedure Flush;

    -- Function:
    --   This routine insures that all previously written data has
    --   reached the destination device.
  procedure Write(
      data_access: iMAX_Definitions.any_ds;  -- access for segment
                                             -- containing buffer
      offset:      xfer_range;   -- offset of buffer within segment
      length:      xfer_range);  -- number of bytes to transfer

    -- Function:
    --   This routine does a device-dependent write operation.  If
    --   fixed_length is true, the length must be equal to the
    --   preferred buffer length (buffer_length plus one)
    --   For some interfaces, a multiple of the preferred buffer
    --   length may be allowed.


end Sink;



package type Store is

  -- Function:
  --   This package type defines synchronous store.


  -- The following record and three procedures are described in the
  -- Basic_IO_Interface package.

  function Interface_description
    return query_record;

  procedure Close;

  procedure Reset;

  function Transform_interface(
      new_interface_type:  type_description)  -- type of the new
                                              -- interface
    return dynamic_typed;

  function Get_asynchronous_interface
    return Asynchronous_IO_Interface.connection;

  -- end of basic package

  procedure Flush;

    -- Function:
    --   This routine insures that all previously written data has
    --   reached the destination device.
    procedure Read(
        data_access:         iMAX_Definitions.any_ds;
        offset:              xfer_range;
        requested_length:    xfer_range;
        returned_length: out xfer_range);

      -- Function:
      --    This routine does a device-dependent read operation,
      --    controlled by the interface characteristics buffer_length
      --    and fixed_length.
      --
      --    The number of bytes returned will be less than or equal to
      --    that requested when fixed_length is false.  The end_of_file
      --    exception is raised when there is no more data
      --    (returned_length is zero.)
      --
      --    The number of bytes returned is a multiple of the preferred
      --    buffer length (buffer_length plus one) when fixed_length is
      --    true.  The operation_not_allowed exception is raised if the
      --    requested_length is not at least as large as the preferred
      --    buffer length.  The end_of_file exception is raised when
      --    there is no more data (returned_length is zero.)

    procedure Write(
        data_access: iMAX_Definitions.any_ds;
        offset:      xfer_range;
        length:      xfer_range);

      -- Function:
      --    This routine does a device-dependent write operation.  If
      --    fixed_length is true, the length must be equal to the
      --    preferred buffer length (buffer_length plus one).  For some
      --    interfaces, a multiple of the preferred buffer length
      --    may be allowed.

  end Store;

end Synchronous_IO_Interfaces;
```

## TERMINAL INTERFACES PACKAGE

```
with iMAX_Definitions, IO_Definitions, Asynchronous_IO_Interface,
     Synchronous_IO_Interfaces, Unchecked_Conversion;
package Terminal_Interfaces is

  -- Function:
  --   This package includes definitions of the standard synchronous
  --   terminal interfaces and related types.


  use Asynchronous_IO_Interface, Synchronous_IO_Interfaces;

  type terminal_speed is (baud150,  baud300,  baud600, baud1200,
                          baud2400, baud4800, baud9600, baud19200);

  type terminal_status is
    record
      terminal_connected: boolean;  -- terminal is operational if true
    end record;

  type terminal_source_characteristics is
    record
      status: terminal_status;  -- terminal status information
      speed:  terminal_speed;   -- terminal line baud rate
    end record;

  type terminal_sink_characteristics is
    record
      status: terminal_status;  -- terminal status information
      speed:  terminal_speed;   -- terminal line baud rate
    end record;
```

```
package type Terminal_Source is

  -- Function:
  --   This package type defines common terminal input interface.

  -- The following record and three procedures are described in the
  -- Basic_IO_Interface package.

  function Interface_description
    return IO_Definitions.query_record;

  procedure Close;

  procedure Reset;

  function Transform_interface(
      new_interface_type: IO_Definitions.source_transform_types)
    return any_access;
  --  See note in description of Transform_interface in
  --  Synchronous_IO_Interfaces.
  --        new_interface_type:  type_description)
  --      return dynamic_typed;

  function Get_asynchronous_interface
    return Asynchronous_IO_Interface.connection;

  -- end of basic package

  procedure Read(
      data_access:          iMAX_Definitions.any_ds;
                              -- access for segment containing buffer
      offset:               xfer_range;
                              --offset of buffer within segment
      requested_length:     xfer_range;
                              -- number of bytes to read
      returned_length: out xfer_range);
                              -- number of bytes actually read

    -- Function:
    --   This routine does a device-dependent read operation.
    --   The number of bytes returned will be less than or equal
    --   to the number of bytes requested.
    --   A "Read" operation will typically return all characters
    --   up to and including a newline character.
    function Get_terminal_characteristics
      return terminal_source_characteristics;

      -- Function:
      --   This routine allows program to determine the terminal
      --   characteristics.

    procedure Set_terminal_characteristics(
        new_chars: terminal_source_characteristics);

      -- Function:
      --   This routine causes terminal characteristics to be set
      --   as specified in "terminal_source_characteristics" record.  If
      --   only some of characteristics are to be changed, the
      --   safest procedure is to use Get_terminal_characteristics to
      --   copy the structure, change the desired entries in the copy,
      --   then call Set_terminal_characteristics.

  end Terminal_Source;
```

```
package type Terminal_Sink is

  -- Function:
  --   This package type defines common terminal output interface.

  -- The following record and three procedures are described in the
  -- Basic_IO_Interface package.

  function Interface_description
    return IO_Definitions.query_record;

  procedure Close;

  procedure Reset;

  function Transform_interface(
      new_interface_type: IO_Definitions.sink_transform_types)
    return any_access;
  --  See note in description of Transform_interface in
  --  Synchronous_IO_Interfaces.
  --        new_interface_type:  type_description
  --      return dynamic_typed;

  function Get_asynchronous_interface
    return Asynchronous_IO_Interface.connection;

  -- end of basic package

  procedure Flush;

    -- Function:
    --   This routine insures that all previously written data has
    --   reached the destination device.

  procedure Write(
      data_access: iMAX_Definitions.any_ds;
                       -- access for segment containing buffer
      offset:      xfer_range;   -- offset to buffer within segment
      length:      xfer_range);  -- number of bytes to write

    -- Function:
    --   This routine does a device dependent write operation.
    --   If the device is characterized as fixed_length, then
    --   requests must be exactly the preferred buffer length
    --   (buffer_length plus one).
    function Get_terminal_characteristics
      return terminal_sink_characteristics;

      -- Function:
      --   This routine allows program to determine the terminal
      --   characteristics.

    procedure Set_terminal_characteristics(
        new_chars: terminal_sink_characteristics);

      -- Function:
      --   This routine causes terminal characteristics to be set
      --   as specified in "terminal_source_characteristics" record.  If
      --   only some of characteristics are to be changed, the
      --   safest procedure is to use Get_terminal_characteristics to
      --   copy the structure, change the desired entries in the copy,
      --   then call Set_terminal_characteristics.

  end Terminal_Sink;


  -- Unchecked Conversion routines to help in the current temporary
  -- version of Transform_interface.

  function Any_access_to_Source is new Unchecked_Conversion(
                                         source => any_access,
                                         target => Source);

  function Any_access_to_Terminal_Source is new Unchecked_Conversion(
                                         source => any_access,
                                         target => Terminal_Source);

  function Any_access_to_Sink is new Unchecked_Conversion(
                                         source => any_access,
                                         target => Sink);

  function Any_access_to_Terminal_Sink is new Unchecked_Conversion(
                                         source => any_access,
                                         target => Terminal_Sink);


end Terminal_Interfaces;
```

DEBUG SOURCE PACKAGE

```
with IO_Definitions, Asynchronous_IO_Interface,
     Synchronous_IO_Interfaces, iMAX_Definitions;
package Debug_Source is

  -- Function:
  --   This package type defines an input interface using
  --   debugger I/O.  It parallels the synchronous source
  --   interface defined in Synchronous_IO__Interfaces for program
  --   compatibility, but none of the routines except Read actually
  --   perform any function.


  function Interface_description
    return IO_Definitions.query_record;

  procedure Close;

  procedure Reset;

  function Transform_interface(
      new_interface_type:  IO_Definitions.source_transform_types)
    return any_access;

  function Get_asynchronous_interface
    return Asynchronous_IO_Interface.connection;

  -- end of basic package

  procedure Read(
      data_access:          iMAX_Definitions.any_ds;
      offset:               Synchronous_IO_Interfaces.xfer_range;
      requested_length:     Synchronous_IO_Interfaces.xfer_range;
      returned_length: out  Synchronous_IO_Interfaces.xfer_range);

    -- Function:
    --   This routine does a device dependent read operation.
    --   The number of bytes returned will be less than or equal
    --   to the number of bytes requested.
    --   A "Read" operation will typically return all characters
    --   up to a newline character.

end Debug_Source;
```

## DEBUG SINK PACKAGE

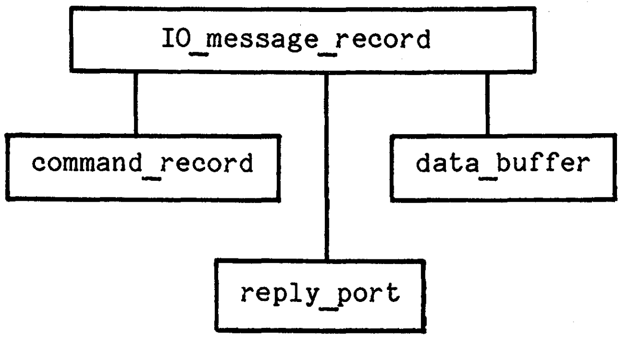
```
with IO_Definitions, Asynchronous_IO_Interface,
     Synchronous_IO_Interfaces, iMAX_Definitions;
package Debug_Sink is

  -- Function:
  --   This package type defines an output interface using
  --   debugger I/O.  It parallels the synchronous sink
  --   interface defined in Synchronous_IO_Interfaces for program
  --   compatibility, but none of the routines except Write actually
  --   perform any function.


  function Interface_description
    return IO_Definitions.query_record;

  procedure Close;

  procedure Reset;

  function Transform_interface(
      new_interface_type:  IO_Definitions.sink_transform_types)
    return any_access;

  function Get_asynchronous_interface
    return Asynchronous_IO_Interface.connection;

  -- end of basic package

  procedure Flush;

    -- Logic:
    --   The debugger I/O interface runs completely synchronously,
    --   so there is never any unwritten data to be flushed.


  procedure Write(
      data_access: iMAX_Definitions.any_ds;
      offset:      Synchronous_IO_Interfaces.xfer_range;
      length:      Synchronous_IO_Interfaces.xfer_range);

    -- Function:
    --   This routine does a device dependent write operation.
    --   If the device is characterized as "fixed_block", then
    --   requests must be exactly "block_size" bytes in length.

end Debug_Sink;
```

## IO DEVICES PACKAGE

```
with Asynchronous_IO_Interface, Synchronous_IO_Interfaces,
     Terminal_Interfaces;
package IO_Devices is

  -- Function:
  --   This package contains the system provided terminal sources and
  --   sinks for iMAX V2.


  subtype interface_range is short_ordinal range 0 .. 5;

  type terminal_device_rep is
    record
      source: Synchronous_IO_Interfaces.Source;
      sink:   Synchronous_IO_Interfaces.Sink;
      tsrc:   Terminal_Interfaces.Terminal_Source;
      tsink:  Terminal_Interfaces.Terminal_Sink;
    end record;

  type terminal_device_list_rep is array (interface_range) of
      terminal_device_rep;

  type terminal_device_list is access terminal_device_list_rep;

  terminals: constant terminal_device_list := new
                          terminal_device_list_rep;


  -- The console interface provides a destination for system messages.

  console: Synchronous_IO_Interfaces.Sink;


  -- The debugger source and sink provide access to the debugger I/O
  -- interface described in the DEBUG-432 EPS.

  debug_source: Synchronous_IO_Interfaces.Source;
  debug_sink:   Synchronous_IO_Interfaces.Sink;


end IO_Devices;
```

## ASYNCHRONOUS IO INTERFACE

```
with IO_Definitions, iMAX_Definitions;
package Asynchronous_IO_Interface is

  -- Function:
  --   This package defines the asynchronous I/O protocol.  This
  --   protocol is used between GDP processes and AP device drivers as
  --   well as within the GDP.


  use IO_Definitions, iMAX_Definitions;

  -- The "connection" type, which is the primary type provided by the
  -- Asynchronous_IO_Interface package, is an access to the following
  -- record type.  All Asynchronous_IO_Interface operators take a
  -- connection as one parameter.

  type connection_record is
    record
      request_port:       port;         -- port for I/O request msgs.
      name:               print_name;   -- identifying name of
                                        -- connection
      device_description: query_record; -- device-specfic info.
      reply_port:         port;         -- port which may be used
                                        -- as a msg. reply port
    end record;

  type connection is access connection_record;

  -- The representation of an I/O transaction is an access segment with
  -- at least three access descriptors.  The first entry is for the
  -- command_record, which is a data segment describing the operation
  -- to be performed.  The second entry is a reply port for the
  -- response message, and the third and succeeding entries are for
  -- data buffers.  Since Ada requires that this structure be declared
  -- in reverse order, here is a picture of what is coming:
  --
  --
  --
  --
  --
  --
  --
  --
  --
  --
  --
  --
```



```
type command_value is range 0 .. short_ordinal'last;
type error_value   is range 0 .. short_ordinal'last;

subtype buffer_range is short_ordinal range 1 .. 1;

type buffer_description is
  record
    offset:           short_ordinal;
    requested_length: short_ordinal;
    returned_length:  short_ordinal;
  end record;

type command_record_rep_val is
  record
    command:             command_value;
    message_id:          short_ordinal;
    reply_code:          error_value;
    buffer_descriptions: array (buffer_range) of buffer_description;
  end record;

type command_record_rep is access command_record_rep_val;

type buffer_array is array (buffer_range) of any_access;

-- I/O command codes
reset:                          constant command_value := 1;
read:                           constant command_value := 2;
write:                          constant command_value := 3;
close:                          constant command_value := 4;
flush:                          constant command_value := 5;
set_device_characteristics:     constant command_value := 6;
get_device_characteristics:     constant command_value := 7;

-- reply codes
success:              constant error_value := 0;
end_of_file:          constant error_value := 1;
not_processed:        constant error_value := 2;
reset_required:       constant error_value := 3;
invalid_command:      constant error_value := 4;
bad_data_buffer_size: constant error_value := 5;
invalid_request:      constant error_value := 6;
hard_IO_error:        constant error_value := 7;
interface_closed:     constant error_value := 8;
reset_returned:       constant error_value := 9;
```

```
  type IO_message_record is
    record
      command_record:     command_record_rep;
      reply_port:         port;
      data_buffer:        buffer_array;
    end record;

  type IO_message is access IO_message_record;


  procedure Send(
      c:   connection;
      msg: IO_message);

    -- Function:
    --   This routine sends a message to an I/O device using
    --   the request_port in the connection.

  procedure Receive(
      c:        connection;
      msg: out IO_message);

    -- Function:
    --   This routine receives a reply to an I/O request on the
    --   reply_port in the connection.

  procedure Cond_send(
      c:            connection;
      msg:          IO_message;
      success: out boolean);

    -- Function:
    --   This routine attempts to send an I/O request message to
    --   the request_port in the connection.  If the operation succeeds,
    --   success is assigned true.  If the port is full and the
    --   operation fails, success is assigned false and the request
    --   message is not sent.

  procedure Cond_receive(
      c:            connection;
      msg:     out IO_message;
      success: out boolean);

    -- Function:
    --   This routine attempts to receive a reply to an I/O request
    --   using a conditional receive on the reply_port in the
    --   connection.

end Asynchronous_IO_Interface;
```

# CHAPTER IOI
# INPUT/OUTPUT IMPLEMENTATION

This chapter describes the implementation of iMAX I/O. It is not directed at general users of iMAX but at system implementers who are modifying iMAX I/O software (e.g., adding a device driver). The reader is presumed to be familiar with the material in these manuals:

- PL/M-86 User's Guide
- iRMX 88 Reference Manual
- iRMX 80/88 Interactive Configuration Utility User's Guide
- iAPX 432 Interface Processor Architecture Reference Manual

The reader should also be familiar with the central system interface to iMAX I/O described in Chapter IO, Input/Output.

Aspects of system initialization that only involve iMAX I/O software are covered in this chapter, rather than in Chapter INI, Initialization.

This chapter has these major parts:

- Overview -- of the iMAX I/O implementation model
- Package Descriptions -- of the three major software components in the iMAX I/O implementation: the IP Controller packages, the AP initialization packages, and the central system I/O initialization packages.
- How to Add a Device to iMAX
- How to Add a Peripheral Subsystem to iMAX
- Package Listings

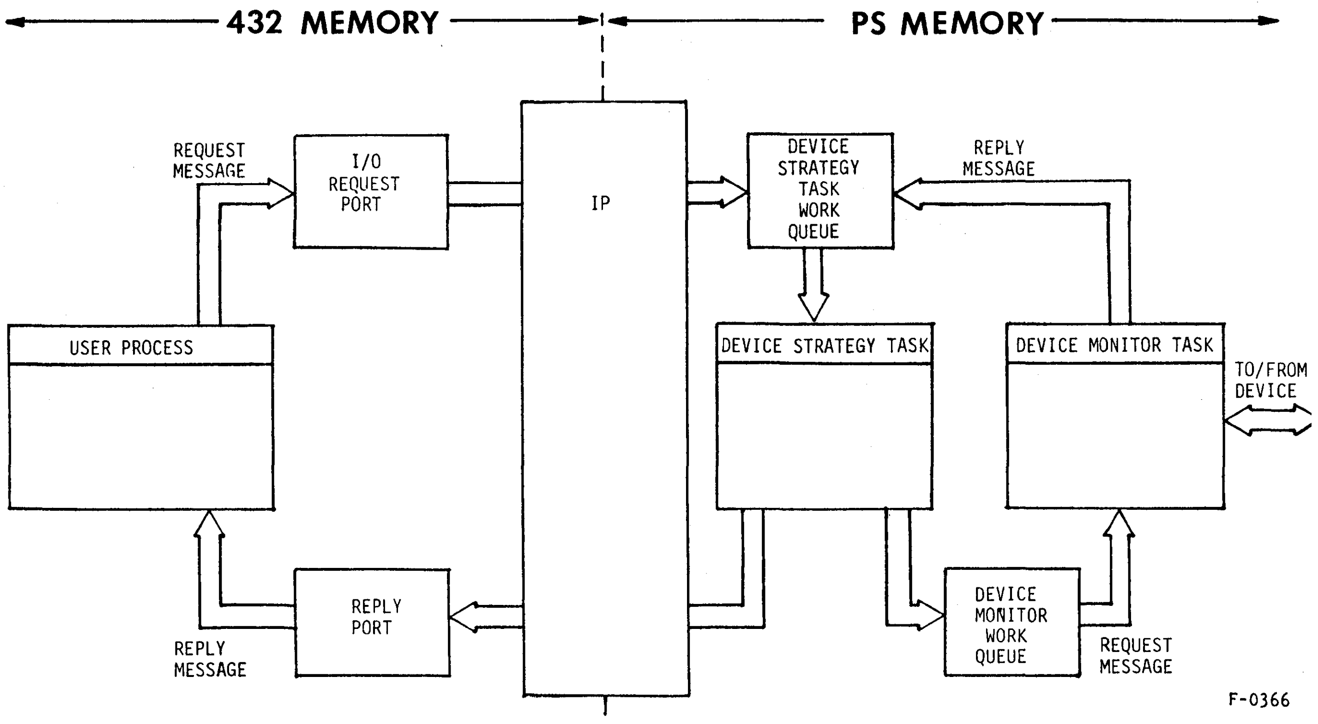


Figure IOI-1 Input/Output Flow

## OVERVIEW

Figure IOI-1 illustrates the iMAX I/O implementation model. These are some important features of the model:

- iMAX I/O uses two sets of cooperating software. One set of modules is written in Ada and executes in the 432 central system. The other set of modules is (typically) written in PL/M-86 and runs on the AP. The AP modules rely on a separate AP executive to provide operating system functions on the AP. The AP executive provides tasks (processes), exchanges (ports), messages, and dynamic storage allocation and deallocation.

- The AP software has these parts:

  IP controller software to manage the IP (and to provide an interface to the AP executive).

  device monitor tasks to manage the devices (typically one per device)

  device strategy tasks (typically one per device) to manage message traffic to and from both the device monitor task and the 432 central system

  initialization, to get it all started

- The device interface is represented by a connection object in the central system (which provides access to the request port at which the IP processes wait). The request port is serviced by one or more IP processes. The IP processes are managed by the strategy task for the device.

## PACKAGE DESCRIPTIONS

This section describes the three major visible software components in the iMAX I/O implementation:

| | |
|---|---|
| • the IP controller | iMAX software to control the IP |
| • AP initialization packages | initialize AP hardware, the IP controller, and user device monitor and strategy tasks |
| • central system I/O initialization | create connections and IP processes; start-up IPs in logical mode |

Two other major components, a strategy task and a monitor task, are needed by users adding device drivers and are described in the section How to Add a Device Driver.

## IP CONTROLLER

The iMAX AP software to control the IP is called the IP Controller and consists of six packages:

| | |
|---|---|
| • AP_Executive_Calls | is a standard interface to AP operating system services. By rewriting the body of AP_Executive Calls, Intel or an iMAX user can transport AP software to a different AP operating system (e.g., from iRMX 88 to iRMX 86). |

| | |
|---|---|
| • IP_Basic_Definitions | declares data types and codes that correspond to IP structures. |
| • IP_Function_Interface | defines procedures corresponding to the IP operators. |
| • IP_Transfer_Driver | defines functions to transfer blocks of data to or from central system memory. |
| • IP_Function_Manager | defines the low-level record-based protocol for presenting requests to the IP (used to implement IP_Function Interface and IP_Transfer_Driver). |
| • IP_Window_Manager | defines procedures to allocate and deallocate windows into central system memory (used to implement IP Transfer_Driver). |

Two of these packages, IP_Function_Manager and IP_Window_Manager, are not generally used by the user who is adding a device driver. These packages are low-level modules used within the IP Controller. They are documented for the benefit of those users who are modifying the IP Controller or who choose to bypass the high-level functions to optimize I/O performance. Other users can skip the descriptions of these two packages. Also, many of the definitions in the IP_Basic_Definitions package concern low-level details not seen by users adding device drivers. The description of the IP_Basic_Definitions package distinguishes those declarations relevant to adding device drivers. Figure IOI-2 illustrates the structure of the IP Controller and distinguishes low-level and user-level modules.

| | | | | |
|---|---|---|---|---|
| user-level: | IP_ Basic_ Definitions | IP_Function_ Interface | IP_Transfer_ Driver | AP_Executive_ Calls |
| low-level: | | IP_Function_ Manager | IP_Window_ Manager | |

Figure IOI-2. User-Level and Low-Level IP Controller Packages

Though not itself part of the IP Controller, the package APINIT defines the procedure Initialize$IP$controller, which is called by the AP initialization task supplied by iMAX.

Before Initialize$IP$controller is called, only those IP controller operations that use the IP in physical mode can be called. The 432 central system must be loaded and started using an IP in physical mode. Loading and starting the central system software can be done by either user AP software or by using DEBUG-432. Both methods are described in chapter INI, Initialization. Loading and starting the 432 central system, even if done by another Peripheral Subsystem, must be done before calling Initialize$IP$controller. Initialize$IP$controller waits for the IP to be placed in logical mode by central system initialization software.

On returning from Initialize$IP$controller, only those IP controller operations that use the IP in logical mode can be called.

## AP Executive Calls

The AP Executive Calls package provides a standard interface to the operating system on the Attached Processor. AP_Executive_Calls is used by all iMAX AP software that invokes AP operating system services. Users adding or modifying device drivers should also use AP_Executive Calls to gain these advantages:

- Users can transport their AP software to different AP operating systems by rewriting only the implementation of AP_Executive_Calls.
- Users do not need to learn the AP operating system.

AP_Executive_Calls defines these data types and constants:

| | |
|---|---|
| • type exchange | AP equivalent of a port, used to transfer AP messages between AP tasks. iMAX assumes that an exchange value is four bytes long. In the RMX 88 implementation, an exchange value is a pointer to the RMX 88 data structure for an exchange. |
| • null$exchange | A constant of type exchange that is the null value for exchanges. |
| • type AP$message | Token for an AP data structure used to carry data to and from AP exchanges. In RMX 88, the message data structure is a header followed by an array of bytes for user data. The function Convert$AP$message$to$pointer takes an AP$message value and returns a pointer to the data area in a message. Note that there is no null$AP$message value defined. |
| • type print$name | Six character array used for task names. |

These operations are provided by AP_Executive_Calls:

| | |
|---|---|
| • AP$get$space | Allocates from zero to 65535 bytes of contiguous memory and returns a pointer to it. If the requested memory is not available, then the calling task waits until it is. |
| • AP$return$space | Returns a block of contiguous memory to the AP executive. The caller passes a pointer to the block being freed. The number of bytes being freed (from zero to 65535) must be in the second double-byte of the block being freed (at offset 2 from the pointer value) for the non-megabyte configuration. For the megabyte configuration, the number of bytes being freed must be in the third double-byte of the block. |
| • AP$create$task | Creates a new AP task (analogous to a 432 process). The caller supplies these parameters: a pointer to a six-character name for the task (cannot be null), a pointer to the first instruction to be executed (cannot be null), the number of bytes needed for the task's stack (from zero to 65535), a pointer to a data segment for the task (can be null), the initial task priority (from zero to 255), an exchange at which the task can receive messages (can be null), and a flag indicating whether the task uses an 8087 Numerical Data Processor. The new task is created and begins executing. No value is returned to the caller. |
| • AP$create$message | Creates a new AP message, given the size of the new message's data part (from zero to 65535 bytes). A token for the new message is returned. To get a pointer to the data part of the message, the function Convert$AP$message$to$pointer must be called. The user should take care to use the token value and not the pointer to the data part in any AP_Executive_Calls. |
| • Convert$AP$Message$ to$pointer | Given an AP executive token for an AP message, return a pointer to the data part of the message. |

|  |  |
|---|---|
| • AP$return$message | Returns an AP$message to the AP executive's free space pool, given a token for the message. |
| • AP$create$exchange | Creates a new exchange and returns an exchange value for it. |
| • AP$return$exchange | Returns an exchange to the AP executive's free space pool. |
| • AP$send$message | Sends an AP$message to an exchange. |
| • AP$receive$message | Receives an AP$message from an exchange. |
| • AP$disable$interrupts | Causes the processor to ignore interrupts until they are again enabled. Any interrupts raised while the processor was ignoring them will still be pending. |
| • AP$enable$interrupts | Causes the processor to acknowledge interrupts. Any interrupts raised while the processor was ignoring them will still be pending. |
| • AP$suspend$self | The calling task stops running. |

## IP Basic Definitions

The IP Basic Definitions package declares data types and codes that correspond to IP structures or to standard conventions for IP usage. The definitions in IP_Basic_Definitions are a mix of user-level and low-level information. This section describes only declarations needed by users adding device drivers. Low-level declarations are described by the comments in the package listing.

IP_Basic_Definitions declares these 432 types or values for the AP: true, false, boolean, null, short$ordinal, ordinal, GDP$physical$ address, access$descriptors, inspection$record (for inspecting ODS), entered$access$segment$index, enterable$access$segment$index, access$segment$index, and access$selector.

IP_Basic_Definitions also defines the function Form$access$selector which, given an EAS index and an access index, returns the corresponding access selector value.

These access selector constants are defined: invalid$acc$sel (all bits one, AD 16383 of EAS 3, likely to not exist and thus likely to fault), acc$sel$for$null$AD (references AD 9 in the IP context access segment, which iMAX makes null, handy if you need a null AD value as a parameter to an IP operator).

## IP Function Interface

The IP Function Interface package defines procedures corresponding to the IP operators. (in contrast to the IP_Function_Manager interface to the IP that requires the user to format opcode and parameters in a function request record).

Every procedure defined by IP_Function_Interface has a common result. The common result is a boolean that is true if the operation faulted, else false. All logical mode procedures also have a common first parameter, the index of the IP process requesting the operation.

The IF_asynchronous_send and IF_asynchronous_receive operators defined by IP_Function_Interface specify an AP message and an AP exchange to which the message is to be sent when the port operator completes. Otherwise, the operations correspond to those specified by the iAPX 432 Interface Processor Architecture Reference Manual (IP ARM). The user should read the description of an operator in the IP ARM before using the corresponding IP__Function__Interface procedure. IP__Function Interface does not document most of the operators, but relies on the IP ARM descriptions. Table IOI-1 tabulates the IP_Function_Interface procedures.

Table IOI-1. IP$Function$Interface procedures

```
  IF$alter$map$and$select$data$seq
 *IF$alter$map$and$select$physical$seg
  IF$copy$access$descriptor
  IF$null$access$descriptor
  IF$amplify$rights
  IF$restrict$rights
  IF$retrieve$public$type$rep
  IF$retrieve$type$rep
  IF$retrieve$type$definition
  IF$inspect$access$descriptor
  IF$inspect$object                -- called INSPECT ACCESS by IP ARM
  IF$lock$object
  IF$unlock$object
  IF$enter$access$seg
  IF$enter$process$globals$access$seg
  IF$set$peripheral$subsystem$mode
 *IF$phy$set$peripheral$subsystem$mode
  IF$send
  IF$receive
**IF$asynchronous$send
**IF$asynchronous$receive
  IF$conditional$send
  IF$conditional$receive
  IF$surrogate$send
  IF$surrogate$receive
  IF$send$to$processor
  IF$broadcast$to$processors
 *IF$send$to$physical$processor
  IF$read$processor$status
 *IF$phy$read$processor$status
  IF$indivisibly$add$short$ordinal
  IF$indivisibly$insert$short$ordinal
  IF$dispatch
```

* = physical mode operator
** = does not exactly correspond to an IP ARM operator.

---

## IP Transfer Driver

The IP_Transfer_Driver package provides functions to transfer blocks of data to or from central system memory. These functions hide the details of window allocation, setup, and deallocations. The Transfer$to$AP function copies a block from central system memory to AP memory. The Transfer$from$AP function copies a block from AP memory to central system memory. Both functions have these parameters:

| | |
|---|---|
| ● environment | index of the IP process requesting the transfer. |
| ● object | access selector in the IP process for the GDP object that contains, or will contain, the data. |
| ● offset | byte offset in the GDP object to the start of the data block. |
| ● length | number of bytes to transfer. |
| ● AP$array$ref | pointer to the AP block that contains, or will contain, the data. |

Both functions return a boolean that is true if the operation failed and false if the operation succeeded.

## IP Function Manager

The IP_Function_Manager package provides a low-level interface to IP functions. Requests are presented to the IP in the form of a function$request$record with these parameters:

| | | |
|---|---|---|
| ● | environment | index of the IP process requesting the function. |
| ● | state | will be written with the completion state of the function. |
| ● | opcode | the IP operation code for the requested function. |
| ● | operands | seven double-bytes. Only those operands required by the requested function need to be filled in. The type and layout of required operands is defined by the IP ARM. |
| ● | result | ten double-bytes to be written with any results of the operation. The type and layout of results is defined by the IP ARM. |
| ● | fault$result | process-level fault code written if the request faults. |
| ● | delayed$response$ port | (only needed for port operations) AP exchange to which message is sent when request completes. |
| | delayed$response$ msg | (only needed for port operations) AP message to be sent to delayed$response$port when request completes. |

IP_Function_Manager defines the type function$request as a pointer to a function$request$record. The operations defined by IP_Function_Manager are passed pointers to function request records rather than the records themselves.

Each IP process has one associated function request record, provided so that the IP_Function_Interface package does not have to dynamically create and destroy the records. The function Get$function$request takes an IP process index and returns a pointer to the function request record for the process. No synchronization is provided for use of these records. Therefore, generally only one AP task should use the function request record for a particular IP process.

The Request$function procedure takes a function request and presents it to the IP. The details of queuing and synchronizing requests and of writing them to the IP's control window are hidden. Control returns when the request is completed and the IP is released.

Either logical or physical mode operations can be requested via Request$function. For physical mode operations, the IP controller must not have been initialized and the environment parameter must be specified as OFFFFH.

The low-level interface to operator faulting associates an exchange with each IP process to which a message is sent if a process-level fault occurs. The message is actually sent after the fault is handled when the process is redispatched. The message sent to the fault exchange is simply a pointer to the function request record that triggered the fault. IP_Function_Manager provides operations to get and to change the fault exchange associated with a particular IP process. Get$fault$exchange takes an IP process index and returns the associated fault exchange. Set$fault$exchange takes an IP process index and an exchange value and makes the exchange value the fault exchange for the IP process. Note that the fault exchange of an IP process should not be null. Note also that the fault exchange associated with an IP process is different than the delayed response exchange defined in the function request record.

IP_Function_Manager also defines the procedure Frmgr$init that must be called to initialize the package before any of the other operations in it are called. The user should not call Frmgr$init directly, but should instead use the Initialize$IP$Controller procedure which initializes all the IP controller modules in the proper sequence.

## IP Window Manager

The IP Window Manager package provides functions Get$window and Return$window to get and return IP windows. Get$window has these parameters:

| | |
|---|---|
| • xfer$mode | whether the window is opened to transfer data from 432 memory to AP memory, or from AP memory to 432 memory. |
| • environment | index of the IP process requesting the window. |
| • object | access selector in the IP process for the GDP object that is to be mapped by the window. |
| • offset | byte offset in the GDP object to the start of the area to be mapped. |
| • length | number of bytes to map minus one. |
| • io$seg | pointer to a description record in which information about the allocated window is placed for use by the caller. This record must be provided by the caller. |

The Return$window function has two parameters:

| | |
|---|---|
| • environment | index of the IP process returning the window. |
| • mapping | pointer to a description record corresponding to the window being deallocated. |

Return$window closes and releases the window and also marks the description record as released.

Both Get$window and Return$window return a boolean that is true if the operation failed and false if the operation succeeded.

The description record used by Get$window and Return$window is referenced by a pointer value of type io$segment. The record contains a pointer to the window, the index (0..3) of the window allocated, and a boolean flag that indicates whether the description record is valid or not. The caller of Get$window is responsible for allocating the description record.

IP_Window_Manager defines the window$faulted array of 5 booleans, one for each IP window (including the control window). Each boolean is true if the corresponding window has faulted, else false.

IP_Window_manager also defines the procedure window$init that must be called to initialize the package before Get$window or Return$window are called. The user should not call window$init directly, but should instead use the Initialize$IP$controller procedure which initializes all the IP controller modules in the proper sequence.

## AP INITIALIZATION

iMAX provides two packages used for AP initialization:

- APINIT provides access to initialization routines for AP software and hardware.
- INITLZ is the user-modifiable initial task for the AP system. It initializes AP software and hardware and then suspends itself.

Each of these packages comes in three versions, for the no terminal, one terminal, and five terminal configurations of the AP software (APINI0, APINI1, APINI5, INITL0, INITL1, and INITL5).

APINIT provides access to these initialization routines:

- Initialize$IP$controller takes two parameters. The first is the index of the last IP process controlled via this IP controller. This value must be greater than or equal to the index of the actual last IP process created for this IP by the IP_Processes package body. The second parameter is a boolean that must be true if AP system error messages are to be written to the terminal on the 86/12A serial port (terminal 1).

  Only the physical mode operators in the IP Controller can be used before Initialize$IP$controller is called. This routine waits for the IP to go into logical mode (done from the central system via the body of IP_Processes) before returning. After the initialization sequence returns from initialize$IP$controller, only the logical mode operators in the IP Controller can be used.

- Initialize$8612$terminal starts the strategy and monitor tasks for a terminal connected to the serial I/O port of the 86/12A board. Initialize$8612$terminal has several parameters, described in the INITL1 package listing.

- Initialize$534$terminal starts the strategy and monitor tasks for four terminals on the serial ports of the 534 board. initialize$534$terminal has several parameters, described in the INITL5 package listing.

Table IOI-2 lists allowed baud rate values when initializing the terminals.

Table IOI-2. Allowed Baud Rate Values

| | |
|---|---|
| 150 | 2400 |
| 300 | 4800 |
| 600 | 9600 |
| 1200 | 19200 |

INITLZ contains the start$init procedure which executes as the initialization task for the AP. start$init takes these actions:

1. Returns enough space (allocated at compile-time) to the AP executive's free space manager to ensure that later dynamic allocation requests succeed.

2. Initializes the IP controller.

3. Initializes the 86/12 terminal driver and 534 terminal drivers, depending on the configuration (default initial baud rate = 2400 baud).

4. ** Insert call to user initialization code here. **

5. Suspends itself and stops executing.

## CENTRAL SYSTEM I/O INITIALIZATION

This section describes the central system initialization of iMAX I/O. More general information about central system initialization is contained in Chapter INI, Initialization.

Four Ada packages are used for initialization:

| | |
|---|---|
| • IP_Processes | This package has a user-modifiable body that contains the I/O initialization code (in the procedure Initialize). IP Processes calls on the services provided by the other three packages. |
| • Actual_Terminal_Sources | Function to create a terminal source package |
| • Actual_Terminal_Sinks | Function to create a terminal sink package |
| • IP_Management | Procedures to create IP processes and to start IPs |

IP_Processes.Initialize contains these steps:

1. Assign the debug_source, debug_sink, and console variables in the package IO_Devices.

2. For each terminal:
   a) Create a request port for the output side.
   b) Create a Terminal__Sink package specifying the request port and the device name (this creates the output connection as well).
   c) Create a Sink package by refining the Terminal__Sink package.
   d) Create the IP processes associated with the output connection. For each one, specify the IP processor ID, the IP process index, the working space needed in the IP context, and the request port.
   e) Create a request port for the input side.
   f) Create a Terminal_Source package, specifying the request port and the device name (this creates the input connection as well).
   g) Create a Source package by refining the Terminal_Source package.
   h) Create the IP processes associated with the input connection. For each one, specify the IP processor ID, the IP process index, the working space needed in the IP context, and the request port.

3. If the flag control_array (start-ip) is not zero, then start the IP with ID given by the constant IP_processes.IP_processor_ID. Note that if the value is zero, no IP is started (the IP could be started from AP code instead).

Users adding code for a new device should insert it just before the code for starting IPs.

## HOW TO ADD A DEVICE TO iMAX

This section describes how to add a new device interface to iMAX. Table IOI-3 is a checklist for users adding a device, and is followed by sections that explain each item on the checklist.

Table IOI-3. Checklist for Adding a New Device

| | Area | Comments | |
|---|---|---|---|
| 1. | central system initialization | Modify body of IP_Processes package | ____ |
| 2. | extending the asynchronous interface | User_supplied Ada package (optional) | ____ |
| 3. | writing the new synchronous interface | User-supplied Ada package (optional) | ____ |
| 4. | writing the new strategy task | User-supplied PL/M-86 package | ____ |
| 5. | writing the new device monitor task | User-supplied PL/M-86 package | ____ |
| 6. | AP initialization | Modify body of INITLZ PL/M-86 package | ____ |
| 7. | AP software configuration | Modify iMAX ICU88 configuration files | ____ |

## CENTRAL SYSTEM INITIALIZATION FOR THE NEW DEVICE

Figure IOI-3 gives an example of central system initialization for a new device. This code should be inserted in the user-modifiable body of the IP_Processes package just before the point where the IP(s) are started.

```
-- User must supply My_Synchronous_Interfaces and My_Devices packages,
-- and reference them in the environment pragma and with clause.
declare
  my_port:  port := Untyped_Ports.Create_port( -- request port
                      number_of_msg_slots,
                      Untyped_Ports.FIFO,
                      SRO);
  -- one IP PGAS is shared by all your IP processes for the
  -- connection.
  my_pglob: IP_process_globals_rep :=
              new IP_process_globals_rep_val(
                request_port => my_port);
  my_number_of_IP_processes constant := 2;
begin

  My_Devices.this_device := My_Synchronous_Interfaces.Create_this_one(
                              my_port);

  for j in 1 .. my_number_of_IP_processes loop
    prcs_no := prcs_no + 1;
    IP_Management.Create_IP_process(
      IP_processor_id,
      prcs_no,
      number_of_work_ADs,
      number_of_work_bytes,
      any_access(my_port));
  end loop;
end;
```

Figure IOI-3. Central System Initialization for a New Device Example

## EXTENDING THE ASYNCHRONOUS INTERFACE

The iMAX Asynchronous_IO_Interface package, described in Chapter IO, Input/Output, defines the form and interpretation of the IO_messages sent to and received from the AP I/O software. The user who is adding a new device may want to define additional command codes or error reply codes. For example, a user implementing a file system would add command codes for open, create, delete, rename, and seek. The file system would also define new error reply codes, such as file_in_use, and file_does_not_exist.

The user should define extensions to the asynchronous interface needed by a particular subsystem in a separate package, like the example package Asynchronous IO Interface File Extensions in Figure IOI-4. A package that uses the extended asynchronous interface (such as the synchronous interface to files) should reference both the iMAX Asynchronous_IO_Interface package and the extensions package, as shown in Figure IOI-5.

The user defining new command codes and error reply codes should use codes 16384 and above, because iMAX uses or reserves codes in the range 0..16383. Failure to comply with this convention could make user programs incompatible with future releases of iMAX.

```
with IO_Definitions, iMAX_Definitions, Asynchronous_IO_Interface;
package Asynchronous_IO_Interface_File_Extensions is

    --   Function:
    --   Define new command codes and error reply codes needed
    --   to implement a file system.

    use IO_Definitions, iMAX_Definitions,
        Asynchronous_IO_Interface;

    --   new command codes (iMax reserves 0 .. 16383)
    open:    constant command_value: = 16384;
    create:  constant command_value: = 16385;
    delete:  constant command_value: = 16386;
    rename:  constant command_value: = 16387;
    seek:    constant command_value: = 16388;

    --   new reply codes (iMAX reserves 0 .. 16383)
    file_in_use:            constant error_value: = 16384;
    file_does_not_exist:    constant error_value: = 16385;

end Asynchronous_Interface_File_Extensions;
```

Figure IOI-4. Example of Extending the Asynchronous Interface

```
    with Asynchronous_IO_Interface,
         Asynchronous_IO_Interface_File_Extensions;
    package Use_Asynchronous_Extensions is

    --   Function:
    --     Give an example of how to reference the asynchronous
    --     interface with extensions.

    use Asynchronous_IO_Interface,
        Asynchronous_IO_Interface_File_Extensions;
    --   makes definitions in both packages directly visible so that
    --   user need not know whether a definition used is from iMAX or
    --   from the extensions.
.
.
.
```

Figure IOI-5. Example of Referencing the Extended Asynchronous Interface

## WRITING THE NEW SYNCHRONOUS INTERFACE

The user should provide a synchronous interface to the new device. No example is given. Chapter IO, Input/Output, provides a complete description of both the standard iMAX synchronous interface, and of the asynchronous interface protocol that is used to implement it.

## WRITING THE NEW STRATEGY TASK

This section describes how the user adding a device driver to iMAX writes the strategy task for the new device. The strategy task provides these functions for the device driver:

- all communications with the 432 central system.
- all control of the IP via iMAX IP controller software, and handling of any errors resulting from use of the IP controller.
- conversion between central system IO messages and user-defined device messages.
- handling of some kinds of errors in IO_messages.

This section contains an example of a strategy task for a generic Sink (an output-only device that supports only the basic operations required by iMAX).

The strategy task manages one or more IP processes (also called environments). Each IP process specifies a request port at which it should wait for an IO_message. The IP process is also used to send the message back to its reply port. The reply port to which an IO_message should be returned is specified by the message itself.

IP processes also provide access environments that can be used to reference 432 objects via access selectors. This is one reason that IP processes are also called environments. The strategy task uses the access environment to reference the request port, the reply port, and the IO_message. Figure IOI-6 illustrates the access paths used by the example strategy task.

The strategy task example follows figure IOI-6.

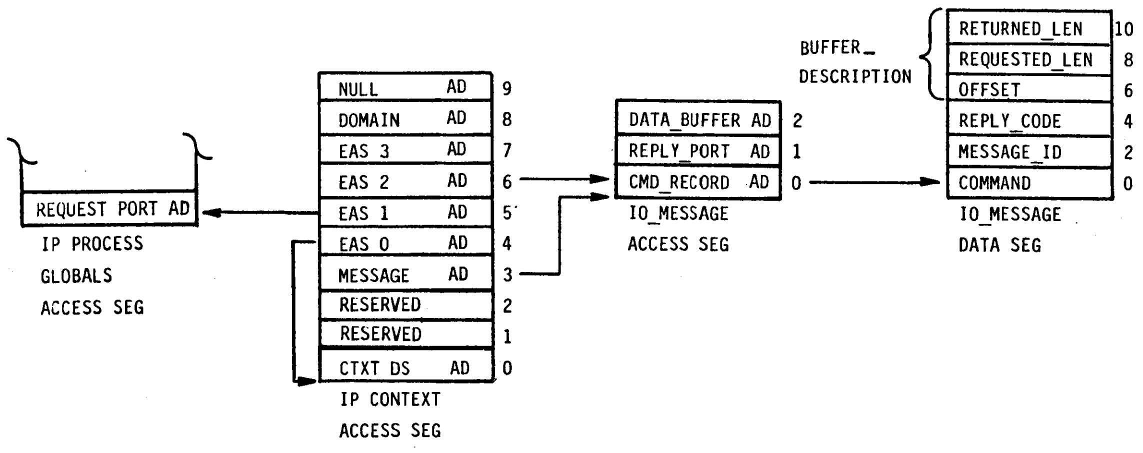


Figure IOI-6. Example Access Path to IO_message

```
/****

Title:   SINKST.PLM - example strategy task procedure for a generic sink

****/

sinkst:  DO;

$INCLUDE (:f1: basdef.inc)  /* use IP_Basic_Definitions,         */
$INCLUDE (:f1: apexec.inc)  /*     AP_Executive_Calls,           */
$INCLUDE (:f1: functn.inc)  /*     IP_Function_Interface,        */
$INCLUDE (:f1: xfer.inc)    /*     IP_Transfer_Driver;           */


/* user-provided error routine is passed a pointer to an error message
   string terminated by a null byte.  The error routine should not
   return control. */
fatal$error:
  PROCEDURE(message$ref) EXTERNAL;
    DECLARE message$ref POINTER;
  END
fatal$error;


sink$strategy:
  PROCEDURE(
    monitor$queue,      /* exchange for ,monitor  task's work       */
    strategy$queue,     /* exchange for strategy task's work        */
    first$env$index,    /* index of first IP process to be managed */
    last$env$index,     /* index of last  IP process to be managed */
    buffer$size)        /* max number of bytes in a data buffer     */
    PUBLIC;
    DECLARE
      monitor$queue   exchange,
      strategy$queue  exchange,
      first$env$index prcs$index,
      last$env$index  prcs$index,
      buffer$size     short$ordinal;


    /* token for message used by AP_Executive_Calls */
    DECLARE device$message$token  AP$message;

    /* pointer to visible part of message */
    DECLARE device$message$ref  POINTER;
DECLARE device$message$state LITERALLY 'short$ordinal';
/* possible values: */
  DECLARE i432$request   LITERALLY '0';
  DECLARE device$request LITERALLY '1';
  DECLARE device$reply   LITERALLY '2';
  DECLARE i432$reply     LITERALLY '3';

  DECLARE last$device$message$state LITERALLY 'i432$reply';

DECLARE device$message BASED device$message$ref STRUCTURE(
  env                prcs$index,
  state              device$message$state,
  command            short$ordinal,
  message$id         short$ordinal,
  reply$code         short$ordinal,
  offset             short$ordinal,
  requested$length   short$ordinal,
  returned$length    short$ordinal,
  buffer             BYTE(0FFFFH)); /* allow for max size; because
                                       structure is BASED, no storage
                                       is allocated by this
                                       declaration. */

DECLARE device$message$size LITERALLY '16 + buffer$size';


/* IO_message encodings (all here, though only some used).  The
   user could put these in a $INCLUDE file (.INC) for inclusion in
   both strategy tasks and monitor tasks. */
DECLARE   reset$cmd                    LITERALLY '1';
DECLARE   read$cmd                     LITERALLY '2';
DECLARE   write$cmd                    LITERALLY '3';
DECLARE   close$cmd                    LITERALLY '4';
DECLARE   flush$cmd                    LITERALLY '5';
DECLARE   set$characteristics$cmd      LITERALLY '6';
DECLARE   get$characteristics$cmd      LITERALLY '7';

DECLARE   success$reply                LITERALLY '0';
DECLARE   end$of$file$reply            LITERALLY '1';
DECLARE   not$processed$reply          LITERALLY '2';
DECLARE   reset$required$reply         LITERALLY '3';
DECLARE   invalid$command$reply        LITERALLY '4';
DECLARE   bad$data$buffer$size$reply   LITERALLY '5';
DECLARE   invalid$request$reply        LITERALLY '6';
DECLARE   hard$IO$error$reply          LITERALLY '7';
DECLARE   interface$closed$reply       LITERALLY '8';
DECLARE   reset$returned$reply         LITERALLY '9';


/* miscellaneous variables */
DECLARE env$index  prcs$index;  /* loop index */

DECLARE faulted boolean;        /* indicates if IP op faulted */
```

```
/* Check parameters. */

IF monitor$queue = null THEN
  CALL fatal$error(@('BAD MONITOR QUEUE PARAMETER',0));
  /* Control does not return.*/

IF strategy$queue = null THEN
  CALL fatal$error(@('BAD STRATEGY QUEUE PARAMETER',0));
  /* Control does not return.*/

IF last$env$index < first$env$index THEN
  CALL fatal$error(@('BAD ENVIRONMENT INDEX PARAMETERS',0));
  /* Control does not return.*/

IF buffer$size = 0 THEN
  CALL fatal$error(@('BAD BUFFER SIZE PARAMETER',0));
  /* Control does not return.*/


DO env$index = first$env$index TO last$env$index;

  faulted = IF$enter$process$globals$access$seg(env$index, 1);
  IF faulted THEN
    CALL fatal$error(@('BAD IP PROCESS',0));
    /* Control does not return. */

  /* The request port for the environment is referenced by AD 0 in
     the process globals access segment (PGAS).  The PGAS is now
     entered as EAS 1 (and should be permanently left as EAS 1).
     form$access$selector(1,0) now references the request port. */

  device$message$token = AP$Create$message(device$message$size);
  device$message$ref = Convert$AP$message$to$pointer(
                           device$message$token);
  /* Each IP process has a device message bound to it in one-to-one
     correspondence. */
  device$message.env = env$index;
  device$message.state = i432$request;

  faulted = IF$asynchronous$receive(env$index,
              device$message$token, strategy$queue,
              form$access$selector(1,0));
  IF faulted THEN
    CALL fatal$error(@('BAD IF$ASYNCHRONOUS$RECEIVE',0));
    /* Control does not return. */

  /* The IP process designated by env$index waits at its request
     port for an IO_message.  When it receives a message, the iMAX
     IP Controller software sends the device$message to the
     strategy$queue exchange.  The strategy task is able to
     manage multiple IP processes waiting at different request
     ports because the asynchronous receive mechanism merges
     the resulting work into a single queue. */
END;
```

```
DO forever;
  device$message$token = AP$receive$message(strategy$queue);
  device$message$ref = Convert$AP$message$to$pointer(
                         device$message$token);

  IF device$message.state > last$device$message$state THEN
    CALL fatal$error(@('BAD DEVICE MESSAGE STATE',0));
    /* Control does not return. */

  ELSE DO CASE device$message.state;
    DO;  /* state = i432$request -- signals that the associated IP
                                     process has received an IO_
                                     message from the request port.*/

      faulted = IF$enter$access$seg(
                  device$message.env,
                  form$access$selector(0, msg$AD$index),
                  2);  /* Enter the IO_message access segment
                          as EAS 2. */
      IF faulted THEN
        device$message.reply$code = invalid$request$reply;
      ELSE DO;
        /* Transfer the command record from the IO_message. (except
           for the returned_length field which is not used). */
        faulted = transfer$to$AP(
                    device$message.env,
                    form$access$selector(2,0),
                    0,
                    10,
                    @device$message.command);
        IF faulted THEN
          device$message.reply$code = invalid$request$reply;
        ELSE DO;
          device$message.reply$code = success$reply;
          /* will be changed if an error is detected */
          IF device$message.command = write$cmd THEN DO;
            IF device$message.requested$length >= buffer$size THEN
              device$message.reply$code =
                bad$data$buffer$size$reply;
            ELSE DO;
              /* Transfer the data to be written to the device. */
              faulted = transfer$to$AP(
                device$message.env,
                form$access$selector(2,2),
                device$message.offset,
                device$message.requested$length + 1,
                @device$message.buffer);
              IF faulted THEN
                device$message.reply$code = invalid$request$reply;
            END;
          END;
        END;
      END;
      device$message.state = device$request;
      CALL AP$send$message(monitor$queue, device$message$token);
      /* Even bad messages are sent through the monitor task to
         preserve message order. */
    END;

    DO; /* state = device$request -- This state should occur only in
                                     messages received by the
                                     monitor task, not in messages
                                     received by the strategy
                                     task. */
      CALL fatal$error(@('BAD DEVICE MESSAGE STATE',0));
      /* Control does not return. */

    END;

    DO; /* state = device$reply -- The device monitor task has
                                   replied.  Now write back the
                                   reply code to the IO_message
                                   and then reply to the 432. */
      faulted = transfer$from$AP(
                  device$message.env,
                  form$access$selector(2,0),
                  4,
                  2,
                  @device$message.reply$code);
      IF faulted THEN
        CALL fatal_error(@('BAD TRANSFER_FROM_AP',0));
        /* Control does not return. */

    /* NOTE: As an optimization, the reply code could be written
             using the IF$indivisibly$insert$short$ordinal
             procedure defined by IP_Function_Interface, which is
             faster than the transfer$from$AP procedure. */

      ELSE DO;
        device$message.state = i432$reply;

        /* Send the IO_message to the 432 reply port specified in
           in the IO_message.  When the send operation completes,
           the device message is sent to the strategy$queue
           exchange. */
        faulted = IF$asynchronous$send(
          device$message.env,
          device$message.token,
          strategy$queue,
          form$access$selector(2, 1),
          form$access$selector(0, msg$AD$index);
        IF faulted THEN
          CALL fatal$error(@('BAD IF$ASYNCHRONOUS$SEND',0));
          /* Control does not return. */

      END;
    END;
    /* NOTE: An alternative to the code for the preceding case is to
             insist that the reply port never block, and use a
             conditional send operation.  If the conditional send
             fails, either raise an error or just drop the
             IO message.  This alternate design has the advantage of
             not tying up IP processes blocked on a reply port, but
             requires careful design to ensure that the reply port
             really doesn't fill up.  If the alternate design is
             used, the code for the last two cases is merged (and
             modified), and there are only three states of device
             messages instead of four. */


    DO; /* state = i432$reply -- The IP process has sent the reply
                                 for the previous IO_message.  The
                                 IP process is now available to
                                 receive another request. */
      device$message.state = i432$request;
      faulted = IF$asynchronous$receive(
                  device$message.env,
                  device$message$token,
                  strategy$queue,
                  form$access$selector(2,0));
      IF faulted THEN
        CALL fatal$error(@('BAD IF$ASYNCHRONOUS$RECEIVE',0));
        /* Control does not return. */

    END;
   END; /* CASE */
  END; /* DO forever */
 END sink$strategy;
END sinkst;
```

## WRITING THE NEW DEVICE MONITOR TASK

This section briefly describes how to write a device monitor task. No example is given. The description given presumes that the device monitor task will communicate with a device strategy task like the one described in the section Writing the New Device Strategy Task.

The monitor task hides all details of the physical device, including registers, timing, and interrupt handling. The monitor task receives I/O requests via an exchange from a strategy task. The monitor task sends I/O replies via an exchange to the strategy task. Both the requests and replies use the same device_message received from the strategy task.

Among the parameters to the monitor task procedure should be the two exchanges, monitor_queue and strategy_queue.

The monitor task loops performing these steps:

1. Receive a device_message from the monitor_queue exchange.

2. a) If the reply_code field is not success_reply, do nothing with the message. This response indicates that the strategy task found an error in the corresponding IO message and has already set the reply_code. The device mesage is just being sent to and from the monitor task to preserve the order in which device_messages are handled.

   b) Otherwise (the reply_code field is success_reply), attempt to carry out the requested command. Then assign the reply_code field if a value other than success_reply should be returned.

3. In either case, change the state field of the device message to the value device_reply, then send the device message to the strategy_queue exchange (back to the strategy task). Then go back to step 1.

Note that the only fields of the device_message that should be modified by the monitor task are the state and reply_code fields (reply_code is set only if the strategy task did not already detect an error and then the monitor task did detect an error).

An alternative way to interface the device monitor task and the strategy task is to add a field to each device_message that specifies the exchange to which it should be returned. This eliminates the strategy_queue parameter and allows the monitor task to be used by multiple requesting tasks. Some of the requesting tasks may not even be associated with 432 I/O requests, but rather may be running independently on the Attached Processor.

## AP INITIALIZATION FOR THE NEW DEVICE

The user should write a separate routine to provide AP initialization for the new device (e.g., initialize$my$device). This routine invokes AP_Executive_Calls to create the exchanges and tasks for the new device.

The user should then add a call to the new initialization routine in the INITLZ module, e.g.:

```
    /* This is where user should insert their own code. */
    CALL initialize$my$device;  /* line inserted by user */

    CALL AP$suspend$self;

  END
start$init;
```

## AP SOFTWARE CONFIGURATION FOR THE NEW DEVICE

AP software is configured using Intel's iRMX 80/88 Interactive Configuration Utility, described in the manual: iRMX 80/88 Interactive Configuration Utility User's Guide.

Appendix SUM, Software Components Summary, lists the Intel-supplied configuration files.

The user must modify the Intel-supplied configuration to include the new strategy task, the new monitor task, and any interrupt handler for the device.

## HOW TO ADD A PERIPHERAL SUBSYSTEM TO iMAX

Adding additional Peripheral Subsystems to iMAX is easy. Do these things:

1. Find out the processor ID number for the new IP. For 432/600 systems, the processor ID is determined by bus slot, as described in Chapter CON, Configuration.

2. Pick a unique short_ordinal identifier for the AP in the new Peripheral Subsystem. Any devices that are located in the new PS should specify this AP__number in their query__record (defined in Chapter IO, Input/Output).

3. Any IP processes created for devices in the new PS must specify the processor ID for the new IP. IP processes are created in the user-modifiable body of the IP__processes. Initialize procedure, described in this chapter.

4. The new IP must be started up in logical mode. This step should be done at the end of the user-modifiable body of the IP_processes.Initialize procedure, described in this chapter. If my_IP_ID is the processor ID of the IP in the new PS, then just add this statement at the end of the IP_Processes. Initialize procedure:

```
IP_Management_Start_IP(my_IP_ID);
```

5. Provide the iMAX software and whatever device drivers you need on the AP side of the new PS. You will also have to ensure that the new PS's AP software is somehow signalled to initialize, if it is not the subsystem that loads and starts the whole system.

## PACKAGE LISTINGS

This section contains listings of all the iMAX AP and central system packages described by this chapter. All AP packages are written in PL/M-86, sometimes with comments that give the equivalent Ada code. Table IOI-4 summarizes the packages and is followed by the listings (in the order shown).

Table IOI-4. Input/Output Implementation
Package Summary

IP Controller Packages (AP)

- AP_Executive_Calls
- IP_Basic_Definitions
- IP_Function_Interface
- IP_Transfer_Driver
- IP_Function_Manager
- IP_Window_Manager

AP Initialization Packages

- INITL0
- INITL1
- INITL5
- APINI0
- APINI1
- APINI5

Central System I/O Initialization Packages

- IP_Processes (body)
- Actual_Terminal_Sources
- Actual_Terminal_Sinks
- IP_Management

AP_EXECUTIVE_CALLS PACKAGE

```
/* with IP_Basic_Definitions;
/*
/* package AP_Executive_Calls is
/*
/*   -- Function:
/*   --   This package provides a procedural interface to the
/*   --   host executive resident on the AP, including definitions
/*   --   of the required structures.
/*
/*   use IP_Basic_Definitions;
/*
/*   -- The exchange mechanism is the port-like mechanism on the AP
/*   -- used for interprocess communication. The IP Controller makes
/*   -- only one assumption about the type "exchange" - that it is 4
/*   -- bytes long.  In the RMX/88 implementation, an "exchange" is
/*   -- really a POINTER to the data structure used by RMX/88 to
/*   -- implement the exchange mechanism.
/*   type exchange is private;
/*   null_exchange:    constant exchange;
/*     -- the null value for exchanges */

DECLARE exchange           LITERALLY 'POINTER';
DECLARE null$exchange      LITERALLY '0H';
/*
/*   -- AP tasks use the exchange mechanism by sending and receiving
/*   -- an "AP_message" to/from an "exchange". In RMX/88, a.message
/*   -- data structure consists of a header and an array of bytes for
/*   -- user data. Users can get a POINTER to the data area in a message
/*   -- by calling the function Convert_AP_message_to_pointer.
/*
/*   type AP_message is new any_access; */

DECLARE AP$message         LITERALLY 'POINTER';
/*
/*   -- The following type is used to give tasks a name.
/*   type print_name is array(0 .. 5) of byte; */

DECLARE print$name         LITERALLY '(6) BYTE';
```

```
/*   function AP_get_space(
/*       size:      short_ordinal)
/*         -- amount of memory, in bytes, to allocate
/*     return pointer;             -- pointer to allocated memory
/*
/*     -- Function:
/*     --   Memory allocation function to be provided by the AP
/*     --   executive. */

AP$get$space:
  PROCEDURE(size) POINTER EXTERNAL;
    DECLARE size  short$ordinal;
  END
AP$get$space;


/*   procedure AP_return_space(
/*       free_space_ptr:   pointer);  -- pointer to free AP memory
/*
/*     -- Function:
/*     --   Routine returns freed up memory to the AP executive's free
/*     --   space manager. The count of the number of bytes being freed
/*     --   must be a short_ordinal in the second double-byte of this
/*     --   memory. (!! third double-byte for megabyte version !!) */

AP$return$space:
  PROCEDURE(free$space$ptr) EXTERNAL;
    DECLARE free$space$ptr  POINTER;
  END
AP$return$space;


/*   procedure AP_create_task(
/*       task_name:         print_name;    -- 6-byte name of task
/*       IP:                pointer;       -- initial instruction pointer
/*       stack_len:         short_ordinal; -- byte count of task stack
/*       data_seg_ptr:      pointer;       -- pointer to task's data
/*                                         -- segment
/*       priority:          short_ordinal; -- initial task priority
/*       task_exch:         exchange;      -- task exchange
/*       NDP_flag:          byte);          -- 0 => no numerical data
/*                                                processor
/*                                          -- 1 => numerical data
/*                                                processor
/*                                          -- other => undefined results
/*
/*     -- Function:
/*     --   Task creation utility provided by the AP executive. */
```

```
AP$create$task:
  PROCEDURE(task$name$ptr, IP, stack$len, data$seg$ptr, priority,
            task$exch, NDP$flag) EXTERNAL;
    DECLARE task$name$ptr    POINTER,
            IP               POINTER,
            stack$len        short$ordinal,
            data$seg$ptr     POINTER,
            priority         short$ordinal,
            task$exch        exchange,
            NDP$flag         BYTE;
  END
AP$create$task;


/*   function AP_create_message(
/*       size:     short_ordinal)
/*         -- amount of memory, in bytes, to allocate
/*         -- for data part of AP message
/*     return AP_message;           -- pointer to allocated message
/*
/*     -- Function:
/*     --   Message allocation function to be provided by the AP
/*     --   executive.  The message header is not made visible to the
/*     --   user - the pointer returned is to the data part of the
/*     --   message. (!! for portability to other executives, the
/*     --   pointer returned should be treated simply as a token for
/*     --   the message, and Convert_AP_message_to_pointer should be
/*     --   called to get a pointer to the data part of the message !!)
/* */

AP$create$message:
  PROCEDURE(size) AP$message EXTERNAL;
    DECLARE size  short$ordinal;
  END
AP$create$message;


/*   function Convert_AP_message_to_pointer(
/*       AP_msg:   AP_message)
/*     return pointer;
/*
/*     -- Function:
/*     --   Converts an AP_message to a pointer. */

convert$AP$message$to$POINTER:
  PROCEDURE(AP$msg) POINTER EXTERNAL;
    DECLARE AP$msg     AP$message;
  END
convert$AP$message$to$POINTER;
```

```
/*   procedure AP_return_message(
/*       free_msg:   AP_message);
/*
/*     -- Function:
/*     --   Routine to return a free message to the AP executive's free
/*     --   space manager. */

AP$return$message:
  PROCEDURE(free$msg) EXTERNAL;
    DECLARE free$msg   AP$message;
  END
AP$return$message;


/*   function AP_create_exchange
/*     return exchange;            -- newly created exchange
/*
/*     -- Function:
/*     --   AP executive function that dynamically creates an exchange.
*/

AP$create$exchange:
  PROCEDURE exchange EXTERNAL;
  END
AP$create$exchange;


/*   procedure AP_return_exchange(
/*       free_exchange:    exchange);
/*
/*     -- Function:
/*     --   Routine to return a no longer needed exchange to the AP
/*     --   executive. */

AP$return$exchange:
  PROCEDURE(free$exchange) EXTERNAL;
    DECLARE free$exchange    exchange;
  END
AP$return$exchange;


/*   procedure AP_send_message(
/*       destination:      exchange;     -- exchange to send message to
/*       message:          AP_message);  -- message to send
/*
/*     -- Function:
/*     --   AP executive version of the interprocess communication SEND
/*     --   operation. */
```

```
AP$send$message:
  PROCEDURE(destination, message) EXTERNAL;
    DECLARE destination   exchange,
            message       AP$message;
  END
AP$send$message;


/*   function AP_receive_message(
/*       source:           exchange)  -- exchange to receive from
/*     return AP_message;             -- message received
/*
/*     -- Function:
/*     --   AP executive version of the interprocess communication
/*     --   RECEIVE operation. */

AP$receive$message:
  PROCEDURE(source) AP$message EXTERNAL;
    DECLARE source   exchange;
  END
AP$receive$message;


/*   procedure AP_disable_interrupts;
/*
/*     --  Function:
/*     --   Causes the processor to ignore all interrupts until a
/*     --   corresponding AP_enable_interrupts is performed.  Any
/*     --   interrupts that were raised while the processor was
/*     --   ignoring them will still be pending. */

AP$disable$interrupts:
  PROCEDURE EXTERNAL;
  END
AP$disable$interrupts;


/*   procedure AP_enable_interrupts;
/*
/*     -- Function:
/*     --   Re-enables interrupts of the AP processor.  Any interrupts
/*     --   that were raised while the processor was ignoring them will
/*     --   still be pending. */

AP$enable$interrupts:
  PROCEDURE EXTERNAL;
  END
AP$enable$interrupts;
```

```
/*   procedure AP_suspend_self;
/*
/*     -- Function:
/*     --   Calling task is suspended. */

AP$suspend$self:
  PROCEDURE EXTERNAL;
  END
AP$suspend$self;


/* private
/*
/*   type exchange is new any_access;
/*     -- AP interprocess communication mechanism
/*     -- any_access is used as a place holder
/*   null_exchange:   constant exchange := null;
/*     -- the null value for exchanges
/*
/*
/* end AP_Executive_Calls; */
```

## IP_BASIC_DEFINITIONS PACKAGE

```
/* package IP_Basic_Definitions is
/*
/*   -- Function:
/*   --   This package contains definitions and routines used to
/*   --   describe the basic types used in the IP Controller. In
/*   --   addition it defines the IP-specific types for the
/*   --   parameters used in IP_Function_Interface.
/*  */

  /* declarations of miscellaneous common symbols. */

DECLARE true                       LITERALLY 'OFFH',
        false                      LITERALLY '000H',
        boolean                    LITERALLY 'BYTE',
        double$byte$boolean        LITERALLY 'WORD',
        forever                    LITERALLY 'WHILE 1',
        null                       LITERALLY 'OH';

DECLARE short$ordinal  LITERALLY 'WORD';
DECLARE ordinal   LITERALLY 'STRUCTURE(lower WORD, upper WORD)';

/*   type memory_array is array(short_ordinal) of byte;
/*   type pointer is access memory_array;
/*  */

  /* single bit masks */
DECLARE bit$0$mask     LITERALLY '01H',
        bit$1$mask     LITERALLY '02H',
        bit$2$mask     LITERALLY '04H',
        bit$3$mask     LITERALLY '08H',
        bit$4$mask     LITERALLY '010H',
        bit$5$mask     LITERALLY '020H',
        bit$6$mask     LITERALLY '040H',
        bit$7$mask     LITERALLY '080H',
        bit$8$mask     LITERALLY '0100H',
        bit$9$mask     LITERALLY '0200H',
        bit$10$mask    LITERALLY '0400H',
        bit$11$mask    LITERALLY '0800H',
        bit$12$mask    LITERALLY '01000H',
        bit$13$mask    LITERALLY '02000H',
        bit$14$mask    LITERALLY '04000H',
        bit$15$mask    LITERALLY '08000H';
  /* multiple bit masks */
DECLARE bits$10$mask    LITERALLY '03H',
        bits$210$mask   LITERALLY '07H',
        bits$3210$mask  LITERALLY '0FH',
        bits$21$mask    LITERALLY '06H',
        bits$43$mask    LITERALLY '018H',
        bits$54$mask    LITERALLY '030H',
        bits$76$mask    LITERALLY '0C0H';

DECLARE upper$byte$mask LITERALLY '0FF00H';
/*
/*   -- BASIC 432 TYPES NEEDED BY THE AP
/*
/*   type boolean_acc is access boolean;
/*   type short_ordinal_acc is access short_ordinal;
/*   type GDP_physical_address is new ordinal range 0 .. 2**24 - 1;
/*     -- used when IP is in physical mode */

DECLARE GDP$physical$address  LITERALLY 'ordinal';
/*
/*   type access_descriptor is new ordinal;
/*     -- data representation of an access descriptor which is
/*     -- returned by Inspect_access_descriptor
/*   type access_descriptor_acc is access access_descriptor; */

DECLARE access$descriptor     LITERALLY 'ordinal';
/*
/*   type inspection_record is array(1 .. 10) of short_ordinal;
/*     -- data format returned by Inspect_object
/*   type inspection_record_acc is access inspection_record; */

DECLARE inspection$record     LITERALLY
          'STRUCTURE(SO(10) short$ordinal)';
/*
/*   type IPC_message is (select_process,
/*                        start_processor,
/*                        stop_processor,
/*                        set_broadcast_acceptance_mode,
/*                        clear_broadcast_acceptance_mode,
/*                        flush_object_table,
/*                          -- same as flush_object_table_cache on GDP.
/*                        suspend_and_fully_requalify_psor,
/*                        suspend_and_requalify_psor,
/*                        suspend_and_requalify_process,
/*                        invalidate_and_unlock_windows,
/*                        generate_PS_reset,
/*                        set_physical_reference_mode); */
```

```
DECLARE IPC$message  LITERALLY 'WORD';
DECLARE select$process                        LITERALLY '0',
        start$processor                       LITERALLY '1',
        stop$processor                        LITERALLY '2',
        set$broadcast$acceptance$mode         LITERALLY '3',
        clear$broadcast$acceptance$mode       LITERALLY '4',
        flush$object$table                    LITERALLY '5',
        suspend$and$fully$requalify$psor      LITERALLY '6',
        suspend$and$requalify$psor            LITERALLY '7',
        suspend$and$requalify$process         LITERALLY '8',
        invalidate$and$unlock$windows         LITERALLY '15',
        generate$PS$reset                     LITERALLY '16',
        set$physical$reference$mode           LITERALLY '17';
/*
/*   -- In the following field, write-sample-delay is the low order bit
/*   -- and xack-delay is the high order two bits. The four legal
/*   -- combinations are enumerated for user convenience.
/*   type write_sample_delay_and_xack_delay
/*     is (case_1, case_2, case_3, case_4); */

DECLARE case$1         LITERALLY '000B',
        case$2         LITERALLY '001B',
        case$3         LITERALLY '010B',
        case$4         LITERALLY '101B';
/*
/*   type PS_status is
/*     record
/*       WSD_and_XD:                write_sample_delay_and_xack_delay;
/*       interrupt_inhibit:         boolean;
/*     end record; */

DECLARE PS$status          LITERALLY 'WORD';
DECLARE write$sample$delay$and$xack$delay  LITERALLY 'bits$210$mask',
        interrupt$inhibit                  LITERALLY 'bit$3$mask';
```

```
/*   type IP_operator is (-- logical mode operators.
/*
/*                            alter_map_and_select_data_segment,
/*                            copy_access_descriptor,
/*                            null_access_descriptor,
/*                            amplify_rights,
/*                            restrict_rights,
/*                            retrieve_public_type_representation,
/*                            retrieve_type_representation,
/*                            retrieve_type_definition,
/*                            inspect_access_descriptor,
/*                            inspect_object,
/*                            lock_object,
/*                            unlock_object,
/*                            indivisibly_add_short_ordinal,
/*                            indivisibly_insert_short_ordinal,
/*                            enter_access_segment,
/*                            enter_global_access_segment,
/*                            set_PS_mode,
/*                            send,
/*                            receive,
/*                            conditional_send,
/*                            conditional_receive,
/*                            surrogate_send,
/*                            surrogate_receive,
/*                            send_to_processor,
/*                            broadcast_to_processors,
/*                            read_processor_status,
/*                            dispatch,
/*
/*                            -- physical mode operators.
/*
/*                            alter_map_and_select_physical_segment,
/* --                           set_PS_mode,
/*                            send_to_physical_processor --,
/* --                           read_processor_status
/*                           );
/*  */
```

```
DECLARE IP$operator                    LITERALLY 'WORD';
DECLARE alter$map$and$select$data$seg          LITERALLY '0',
        alter$map$and$select$physical$seg      LITERALLY '0',
        send$to$processor                      LITERALLY '1',
        send$to$physical$processor             LITERALLY '1',
        set$PS$mode                            LITERALLY '2',
        read$processor$status                  LITERALLY '3',
        copy$access$descriptor                 LITERALLY '4',
        null$access$descriptor                 LITERALLY '5',
        enter$global$access$segment            LITERALLY '6',
        enter$access$segment                   LITERALLY '7',
        amplify$rights                         LITERALLY '8',
        restrict$rights                        LITERALLY '9',
        retrieve$type$rep                      LITERALLY '10',
        retrieve$public$type$rep               LITERALLY '11',
        retrieve$type$definition               LITERALLY '12',
        /* formerly retrieve$refined$object was 13 */
        inspect$access$descriptor              LITERALLY '14',
        inspect$object                         LITERALLY '15',
        lock$object                            LITERALLY '16',
        unlock$object                          LITERALLY '17',
        send                                   LITERALLY '18',
        conditional$send                       LITERALLY '19',
        receive                                LITERALLY '20',
        conditional$receive                    LITERALLY '21',
        surrogate$send                         LITERALLY '22',
        surrogate$receive                      LITERALLY '23',
        broadcast$to$processors                LITERALLY '24',
        indivisibly$add$short$ordinal          LITERALLY '25',
        indivisibly$insert$short$ordinal       LITERALLY '26',
        dispatch                               LITERALLY '27';
/*
/*        -- NOTE: The process indices used by the IP Controller are:
/*        --             0, 1, 2, ... , n   for n+1 processes.
/*        --
/*        --       The IP microcode expects this index to be shifted
/*        --       left by two bits; but the only time the microcode
/*        --       sees it is in the function request facility. Any
/*        --       process index used in a funtion request will be left
/*        --       shifted two bits to be put in:
/*        --             0, 4, 8, ... , 4n    for n+1 processes.
/*        --
/*   subtype prcs_index
/*     is short_ordinal range 0 .. short_ordinal'last; */

DECLARE prcs$index    LITERALLY 'short$ordinal';
/*
/*   type function_completion_state is (
/*     -- condition of function completion
/*         invalid_value,  -- IP has not yet started executing function
/*         in_progress,    -- function executing but not yet completed
/*         completed);     -- function completed */
```

```
DECLARE completed            LITERALLY '0H',
        in$progress          LITERALLY '01H',
        invalid$value        LITERALLY '0FH';
/*
/*   type fault_level is (
/*        none,
/*        context,
/*        process,
/*        processor); */

DECLARE none        LITERALLY '000H',
        context     LITERALLY '040H',
        process     LITERALLY '080H',
        processor   LITERALLY '0C0H';
/*
/*   type process_fault_reply is new short_ordinal;
/*
/*   type IP_fault_status is (    -- indication from process-level fault
/*                                -- handler as to how fault was handled
/*      context_fault,-- turned into a context-level fault
/*      retry,        -- fault was handled, operation should be retried
/*      resume);      -- fault was handled, operation was completed
/*
/*   type IP_results_rec is
/*     record
/*       fault_result:     IP_fault_status;
/*       cond_op_result:        short_ordinal range 0 .. 255;
/*     end record; */

DECLARE process$fault$reply    LITERALLY 'short$ordinal';

DECLARE IP$fault$status     LITERALLY 'BYTE';
DECLARE context$fault             LITERALLY '3H',
        retry                     LITERALLY '1H',
        resume                    LITERALLY '2H';

DECLARE IP$results$rec  LITERALLY 'STRUCTURE(
  fault$result     IP$fault$status,
  cond$op$result   BYTE)';

/*   type function_state is  -- info on how the function request fared
/*     record
/*       f_completion_state:          function_completion_state;
/*       send_blocked:                boolean;
/*       receive_blocked:             boolean;
/*       fault_level_state:           fault_level;
/*     end record; */

DECLARE function$state    LITERALLY 'WORD';
DECLARE f$completion$state   LITERALLY 'bits$3210$mask',
        send$blocked         LITERALLY 'bit$4$mask',
        receive$blocked      LITERALLY 'bit$5$mask',
        fault$level$state    LITERALLY 'bits$76$mask';
```

```
/*   type processor_state is (
/*        idle,
/*        process_execution); */

DECLARE idle              LITERALLY '0',
        process$execution LITERALLY '1';
/*
/*   type reference_mode is (
/*        logical,
/*        physical); */

DECLARE physical          LITERALLY '000H',
        logical           LITERALLY '020H';
/*
/*    type   processor_status   is    --  only   written,   never   read   by
processors.
/*     record
/*       psor_state:                 processor_state;
/*       faulted:                    boolean;
/*       ref_mode:                   reference_mode;
/*       stopped:                    boolean;
/*         -- true => stopped, false => running by IPC.
/*       broadcast_acceptance_mode:  boolean;
/*         -- true  => accepting broadcast IPCs,
/*         -- false => not accepting broacast IPCs.
/*       psor_id:                    byte;
/*     end record; */

DECLARE processor$status      LITERALLY 'WORD';
DECLARE psor$state                 LITERALLY 'bits$3210$mask',
        psor$faulted               LITERALLY 'bit$4$mask',
        ref$mode                   LITERALLY 'bit$5$mask',
        stopped                    LITERALLY 'bit$6$mask',
        broadcast$acceptance$mode  LITERALLY 'bit$7$mask',
        psor$id                    LITERALLY 'upper$byte$mask';
/*
/*   type processor_status_record is
/*     record
/*       status:   processor_status;
/*       clock:    short_ordinal;
/*     end record;
/*
/*   type processor_status_record_acc
/*     is access processor_status_record; */

DECLARE processor$status$record LITERALLY 'STRUCTURE(
    status    processor$status,
    clock     short$ordinal)';
```

```
/*   -- IP WINDOW DESCRIPTION DEFINITIONS
/*
/*   -- window indices
/*
/*   type window_index is range 0 .. 4; */

DECLARE window$index            LITERALLY 'short$ordinal';
/*
/*   subtype alterable_window_index is window_index range 0 .. 3;
/*     -- window 4 maps onto a refinement of the processor data
/*     -- segment, and cannot be altered in logical mode */
DECLARE alterable$window$index  LITERALLY 'window$index';
/*
/*   type transfer_state is (      -- only meaningful for window 0 when
/*                                 -- set up for a block mode transfer
/*         xfer_in_progress,               -- transfer in progress
/*         termination_upon_count_runout,  -- entire transfer completed
/*         termination_forced,  -- transfer forced to terminate by
/*                              -- alter window function request
/*         termination_upon_fault);  -- transfer terminated by window
/*                                   -- fault */

DECLARE xfer$in$progress                LITERALLY '00H',
        termination$upon$count$runout   LITERALLY '10H',
        termination$forced              LITERALLY '20H',
        termination$upon$fault          LITERALLY '30H';
/*
/*
/*   type window_entry_state is    -- entry information for a window
/*     record
/*       valid:  boolean;
/*       -- true => window is in use
/*
/*       -- The following fields are meaningful only if "valid" is true
/*       -- with one exception: if valid is false, and the window
/*       -- specified as another parameter to the alter-map operation
/*       -- is window 0 which is currently in block mode, then a
/*       -- trf_state field value of termination_forced will cause
/*       -- termination of that transfer without actually invalidating
/*       -- the window.
/*       block_interconnect_transfer_mode:    boolean;
/*         -- true for window 0 => block mode transfer;
/*         -- true for window 1 => interconnect mode transfer;
/*         -- cannot be true for windows 2 - 4
/*       read_allowed:                        boolean;
/*         -- true => data can be read from 432 memory via this mapping
/*       write_allowed:                       boolean;
/*         -- true => data can be written to 432 memory via this mapping
/*       trf_state:                           transfer_state;
/*         -- this field is only meaningful for window 0 when set
/*         -- up in block mode, and is the current transfer state
/*       mem_overlay:                         boolean;
/*         -- true => this mapping overlays real AP addresses
/*     end record; */
```

```
DECLARE window$entry$state    LITERALLY 'WORD';
DECLARE valid                           LITERALLY 'bit$0$mask',
        block$interconnect$transfer$mode LITERALLY 'bit$1$mask',
        read$allowed                    LITERALLY 'bit$2$mask',
        write$allowed                   LITERALLY 'bit$3$mask',
        trf$state                       LITERALLY 'bits$54$mask',
        mem$overlay                     LITERALLY 'bit$6$mask';
/*
/*    subtype   invalidation_state   is   window_entry_state   suchthat
instance.valid = false; */

DECLARE invalidation$state    LITERALLY '0H';
/*
/*   -- ENTERED ACCESS TYPES AND CONSTANTS (INCLUDING ACCESS SELECTORS)
/*
/*   type entered_access_segment_index
/*     is new short_ordinal range 0 .. 3; */

DECLARE entered$access$segment$index LITERALLY 'short$ordinal';
/*
/*   subtype enterable_entered_access_segment_index
/*     is entered_access_segment_index range 1 .. 3;
/*   -- entered access list 0 (the current context) cannot be entered */

DECLARE enterable$entered$access$seg$index
        LITERALLY 'entered$access$segment$index';
/*
/*   type access_segment_index is
/*     new short_ordinal range 0 .. 2**14 - 1;
/*   -- index into an entered access list */

DECLARE access$segment$index   LITERALLY 'short$ordinal';
/*
/*   type access_selector is new short_ordinal;
/*     -- access selectors must be used explicitly by the AP
/*     -- software to access the 432 environments */

DECLARE access$selector        LITERALLY 'short$ordinal';
```

```
/*   function form_access_selector(
/*       entered_AS_index:
/*         entered_access_segment_index;  -- two components of an
/*       AS_index:  access_segment_index) -- access selector
/*     return access_selector;            -- access selector formed
/*
/*     -- Function:
/*     --   This function takes the component values of an access
/*     --   selector and returns an access selector.
/*
/*   invalid_acc_sel:  constant access_selector :=
/*                     form_access_selector(3,((2**14)-1));  -- i.e. -1
/*     -- used as a default parameter */

DECLARE invalid$acc$sel   LITERALLY '01111$1111$1111$11$$11B';
/*
/*   acc_sel_for_null_AD:  constant access_selector :=
/*                         form_access_selector(0,9);
/*     -- used when a null access descriptor is needed as
/*     -- a parameter for a function request */

DECLARE acc$sel$for$null$AD   LITERALLY '01001$00B';
/*
/*   -- The context is always accessible as entered access segment 0.
/*   context_AS:  constant entered_access_segment_index := 0; */

DECLARE context$AS      LITERALLY '0H';
/*
/*   -- The message slot in the context is at offset 3.
/*   msg_AD_index:  constant access_segment_index := 3; */

DECLARE msg$AD$index   LITERALLY '03H';


form$access$selector:
  PROCEDURE(entered$AS$index, AS$index) access$selector EXTERNAL;
    DECLARE entered$AS$index    entered$access$segment$index;
    DECLARE AS$index          access$segment$index;
  END
form$access$selector;
/*
/* end IP_Basic_Definitions; */
```

IP_FUNCTION_INTERFACE PACKAGE

```
with IP_Basic_Definitions, AP_Executive_Calls;
/*
/* package IP_Function_Interface is
/*
/*    -- Function:
/*    --   IP_Function_Interface provides a procedural interface to
/*    --   the IP function request facility. A few of these procedures
/*    --   also specify an AP message and an AP exchange where it is to
/*    --   be sent asynchronously in case the function execution blocks
/*    --   (port operators). All of the routines return a boolean to
/*    --   indicate a context-level fault (true => context-fault).
/*    --
/*    --   All the physical mode and logical mode operators of the IP are
/*    --   provided by this package.  Note that the physical mode
/*    --   operators can only be invoked before the IP Controller is
/*    --   initialized; after initialization only the logical mode
/*    --   operators can be called.
/*    --
/*    --   Note: Most of the procedures in this package are without
/*    --   comments since they are merely procedural interfaces to
/*    --   various IP operators.
/*
/*    use IP_Basic_Definitions, AP_Executive_Calls;
/*
/*
/*    function Alter_map_and_select_data_segment(
/*        environment:       prcs_index;
/*          -- process requesting operation
/*        window_id:         window_index;
/*          -- window to be altered
/*        new_state:         window_entry_state;
/*          -- new state for that window
/*        AP_base:           short_ordinal := 0;
/*          -- start of AP address range to be mapped
/*          --    relative to 64K region mapped by IP
/*        AP_mask:           short_ordinal := 0;
/*          -- mask governing length of AP address range
/*        object:            access_selector := invalid_acc_sel;
/*          -- data segment being mapped
/*        block_count:       short_ordinal := 0;
/*          -- actual length being mapped (bytes)
/*        offset:            short_ordinal := 0)
/*          -- offset into data segment of beginning of area being mapped
/*      return boolean;   -- true if context-faulted
/*
/*      -- Function:
/*      --   This procedure provides a synchronous interface to the IP
/*      --   operator with the same name. It can be used to change the
/*      --   mappings of windows 0 - 3 while the IP is in logical mode.
/*     --
/*     --   If the purpose of the operation is to invalidate a window or
/*     --   to force the termination of the block mode window, the last
/*     --   5 parameters need not be specified. If a window is being set
/*     --   up in random mode, the last 2 parameters need not be
/*     --   specified. */

IF$alter$map$and$select$data$seg:
  PROCEDURE(environment, window$id, new$state,
          AP$base, AP$mask, object, block$count, offset)
      boolean EXTERNAL;
    DECLARE environment     prcs$index,
            window$id       window$index,
            new$state       window$entry$state,
            AP$base         short$ordinal,
            AP$mask         short$ordinal,
            object          access$selector,
            block$count     short$ordinal,
            offset          short$ordinal;
     /* no default values for parameters */
  END
IF$alter$map$and$select$data$seg;


/*    function Alter_map_and_select_physical_segment(
/*        window_id:          alterable_window_index;
/*          -- window to be altered
/*        new_state:          window_entry_state;
/*          -- new state for that window
/*        AP_base:            short_ordinal := 0;
/*          -- start of AP address range to be mapped
/*        AP_mask:            short_ordinal := 0;
/*          -- mask governing length of AP address range
/*        GDP_address:        GDP_physical_address := 0;
/*          -- 432 physical base address of mapped area
/*        block_count:        short_ordinal := 0)
/*          -- actual length being mapped (bytes)
/*      return boolean;     -- true if context-faulted
/*
/*      -- Function:
/*      --   This procedure provides a synchronous interface to the IP
/*      --   operator with the same name. It can be used to change the
/*      --   mappings of windows 0 - 4 while the IP is in physical mode.
/*      --
/*      --   If the purpose of the operation is to invalidate a window or
/*      --   to force the termination of the block mode window, the last
/*      --   4 parameters need not be specified. If a window is being set
/*      --   up in random mode, the last parameter need not be
/*      --   specified. */
```

```
IF$alter$map$and$select$physical$seg:
  PROCEDURE(window$id, new$state, AP$base, AP$mask,
            GDP$addr$lower, GDP$addr$upper, block$count)
      boolean EXTERNAL;
    DECLARE window$id        window$index,
            new$state        window$entry$state,
            AP$base          short$ordinal,
            AP$mask          short$ordinal,
            GDP$addr$lower   short$ordinal,
            GDP$addr$upper   short$ordinal,
            block$count      short$ordinal;
  END
IF$alter$map$and$select$physical$seg;


/*   function Copy_access_descriptor(
/*       environment:      prcs_index;
/*       source:           access_selector;
/*       destination:      access_selector)
/*     return boolean;                -- true if context-faulted */

IF$copy$access$descriptor:
  PROCEDURE(environment, source, destination) boolean EXTERNAL;
    DECLARE environment    prcs$index,
            source         access$selector,
            destination    access$selector;
  END
IF$copy$access$descriptor;


/*   function Null_access_descriptor(
/*       environment:      prcs_index;
/*       destination:      access_selector)
/*     return boolean;                -- true if context-faulted */

IF$null$access$descriptor:
  PROCEDURE(environment, destination) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      destination    access$selector;
  END
IF$null$access$descriptor;


/*   function Amplify_rights(
/*       environment:      prcs_index;
/*       transformer:      access_selector;
/*       destination:      access_selector)
/*     return boolean;                -- true if context-faulted */
```

```
IF$amplify$rights:
  PROCEDURE(environment, transformer, destination) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      transformer    access$selector,
      destination    access$selector;
  END
IF$amplify$rights;


/*   function Restrict_rights(
/*       environment:       prcs_index;
/*       rights_mask:       ordinal;
/*       destination:       access_selector)
/*     return boolean;              -- true if context-faulted */

IF$restrict$rights:
  PROCEDURE(environment, rights$mask$lower, rights$mask$upper,
            destination) boolean EXTERNAL;
    DECLARE
      environment        prcs$index,
      rights$mask$lower  short$ordinal,
      rights$mask$upper  short$ordinal,
      destination        access$selector;
  END
IF$restrict$rights;


/*   function Retrieve_public_type_representation(
/*       environment:       prcs_index;
/*       extended_type:     access_selector;
/*       destination:       access_selector)
/*     return boolean;              -- true if context-faulted */

IF$retrieve$public$type$rep:
  PROCEDURE(environment, extended$type, destination) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      extended$type  access$selector,
      destination    access$selector;
  END
IF$retrieve$public$type$rep;


/*   function Retrieve_type_representation(
/*       environment:       prcs_index;
/*       extended_type:     access_selector;
/*       type_defn:         access_selector;
/*       destination:       access_selector)
/*     return boolean;              -- true if context-faulted */
```

```
IF$retrieve$type$rep:
  PROCEDURE(environment, extended$type, type$defn, destination)
      boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      extended$type  access$selector,
      type$defn      access$selector,
      destination    access$selector;
  END
IF$retrieve$type$rep;


/*   function Retrieve_type_definition(
/*       environment:       prcs_index;
/*       extended_type:     access_selector;
/*       destination:       access_selector)
/*     return boolean;               -- true if context-faulted */

IF$retrieve$type$definition:
  PROCEDURE(environment, extended$type, destination) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      extended$type  access$selector,
      destination    access$selector;
  END
IF$retrieve$type$definition;


/*   function Inspect_access_descriptor(
/*       environment:       prcs_index;
/*       source:            access_selector;
/*       destination:       access_descriptor_acc)
/*     return boolean;               -- true if context-faulted */

IF$inspect$access$descriptor:
  PROCEDURE(environment, source, destination$ptr) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      source         access$selector,
      destination$ptr  POINTER; /* this is a pointer to the
                                   4-byte access$descriptor field
                                   where the result will be put */
  END
IF$inspect$access$descriptor;
```

```
/*   function Inspect_object(
/*       environment:       prcs_index;
/*       source:            access_selector;
/*       destination:       inspection_record_acc)
/*     return boolean;              -- true if context-faulted */

IF$inspect$object:
  PROCEDURE(environment, source, destination$ptr) boolean EXTERNAL;
    DECLARE
      environment       prcs$index,
      source            access$selector,
      destination$ptr   POINTER; /* this is a pointer to the
                                    20-byte inspection$record where
                                    the result will be put */
  END
IF$inspect$object;


/*   function Lock_object(
/*       environment:               prcs_index;
/*       object_to_lock:            access_selector;
/*       lock_displacement:         short_ordinal;
/*         -- displacement into object to lock
/*       success:                   boolean_acc)
/*         -- true if lock successful
/*     return boolean;              -- true if context-faulted */

IF$lock$object:
  PROCEDURE(environment, object$to$lock, lock$displacement,
          success$ptr) boolean EXTERNAL;
    DECLARE
      environment             prcs$index,
      object$to$lock          access$selector,
      lock$displacement       short$ordinal,
      success$ptr             POINTER;
  END
IF$lock$object;


/*   function Unlock_object(
/*       environment:               prcs_index;
/*       object_to_unlock:          access_selector;
/*       unlock_displacement:       short_ordinal)
/*         -- displacement into object to lock
/*     return boolean;              -- true if context-faulted */
```

```
IF$unlock$object:
  PROCEDURE(environment, object$to$unlock, unlock$displacement)
      boolean EXTERNAL;
    DECLARE
      environment           prcs$index,
      object$to$unlock      access$selector,
      unlock$displacement   short$ordinal;
  END
IF$unlock$object;


/*   function Enter_access_segment(
/*       environment:      prcs_index;
/*       source:           access_selector;
/*       entered_AS:       enterable_entered_access_segment_index)
/*     return boolean;             -- true if context-faulted */

IF$enter$access$seg:
  PROCEDURE(environment, source, entered$AS) boolean EXTERNAL;
    DECLARE
      environment   prcs$index,
      source        access$selector,
      entered$AS    enterable$entered$access$seg$index;
  END
IF$enter$access$seg;


/*   function Enter_process_globals_access_segment(
/*       environment:      prcs_index;
/*       entered_AS:       enterable_entered_access_segment_index)
/*     return boolean;             -- true if context-faulted */

IF$enter$process$globals$access$seg:
  PROCEDURE(environment, entered$AS) boolean EXTERNAL;
    DECLARE
      environment   prcs$index,
      entered$AS    enterable$entered$access$seg$index;
  END
IF$enter$process$globals$access$seg;


/*   function Set_peripheral_subsystem_mode(
/*       environment:      prcs_index;
/*       mode_flags:       short_ordinal;
/*       psor_os:          access_selector)
/*     return boolean;             -- true if context-faulted
/*
/*     -- NOTE: This is the logical mode operator only. */
```

```
IF$set$peripheral$subsystem$mode:
  PROCEDURE(environment, mode$flags, psor$os) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      mode$flags     short$ordinal,
      psor$os        access$selector;
  END
IF$set$peripheral$subsystem$mode;


/*   function Phy_set_peripheral_subsystem_mode(
/*       mode_flags:        short_ordinal)
/*     return boolean;              -- true if context-faulted
/*
/*     -- NOTE: This is the physical mode operator only. */

IF$phy$set$peripheral$subsystem$mode:
  PROCEDURE(mode$flags) boolean EXTERNAL;
    DECLARE
      mode$flags     short$ordinal;
  END
IF$phy$set$peripheral$subsystem$mode;


/*   function Send(
/*       environment:       prcs_index;
/*       port:              access_selector;
/*       message:           access_selector)
/*     return boolean;              -- true if context-faulted */

IF$send:
  PROCEDURE(environment, port, message) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      port           access$selector,
      message        access$selector;
  END
IF$send;


/*   function Receive(
/*       environment:       prcs_index;
/*       port:              access_selector)
/*     return boolean;              -- true if context-faulted
/*
/*     -- Function:
/*     --   The received message is left in the message slot of the
/*     --   context access segment of the receiving  IP process. */
```

```
IF$receive:
  PROCEDURE(environment, port) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      port           access$selector;
  END
IF$receive;


/*   function Asynchronous_send(
/*       environment:      prcs_index;
/*       reply_msg:        AP_message;
/*         -- message to be returned when the operation completes
/*       reply_exchange:   exchange;
/*         -- where that message is to be sent
/*       port:             access_selector;
/*       message:          access_selector)
/*     return boolean;         -- true if context-faulted
/*
/*     -- Function:
/*     --   Reply_msg will be sent to reply_exchange when the message is
/*     --   successfully sent. Control returns regardless of whether the
/*     --   operation blocks. */

IF$asynchronous$send:
  PROCEDURE(environment, reply$msg, reply$exchange, port, message)
      boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      reply$msg      AP$message,
      reply$exchange exchange,
      port           access$selector,
      message        access$selector;
  END
IF$asynchronous$send;


/*   function Asynchronous_receive(
/*       environment:      prcs_index;
/*       reply_msg:        AP_message;
/*         -- message to be returned when operation completes
/*       reply_exchange:   exchange;
/*         -- where that message is to be sent
/*       port:             access_selector)
/*     return boolean;              -- true if context-faulted
/*
/*     -- Function:
/*     --   Reply_msg will be sent to reply_exchange when a message is
/*     --   received. Control returns regardless of whether the
/*     --   operation blocks. */
```

```
IF$asynchronous$receive:
  PROCEDURE(environment, reply$msg, reply$exchange, port)
      boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      reply$msg      AP$message,
      reply$exchange exchange,
      port           access$selector;
  END
IF$asynchronous$receive;


/*   function Conditional_send(
/*       environment:        prcs_index;
/*       port:               access_selector;
/*       message:            access_selector;
/*       success:            boolean_acc)
/*     return boolean;               -- true if context-faulted */

IF$conditional$send:
  PROCEDURE(environment, port, message, success$ptr) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      port           access$selector,
      message        access$selector,
      success$ptr    POINTER;
  END
IF$conditional$send;


/*   function Conditional_receive(
/*       environment:        prcs_index;
/*       port:               access_selector;
/*       success:            boolean_acc)
/*     return boolean;               -- true if context-faulted */

IF$conditional$receive:
  PROCEDURE(environment, port, success$ptr) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      port           access$selector,
      success$ptr    POINTER;
  END
IF$conditional$receive;
```

```
/*   function Surrogate_send(
/*       environment:      prcs_index;
/*       port:             access_selector;
/*       destination:      access_selector;
/*       carrier:          access_selector;
/*       message:          access_selector)
/*     return boolean;              -- true if context-faulted */

IF$surrogate$send:
  PROCEDURE(environment, port, destination, carrier, message)
      boolean EXTERNAL;
    DECLARE
      environment   prcs$index,
      port          access$selector,
      destination   access$selector,
      carrier       access$selector,
      message       access$selector;
  END
IF$surrogate$send;


/*   function Surrogate_receive(
/*       environment:      prcs_index;
/*       port:             access_selector;
/*       destination:      access_selector;
/*       carrier:          access_selector)
/*     return boolean;              -- true if context-faulted */

IF$surrogate$receive:
  PROCEDURE(environment, port, destination, carrier) boolean EXTERNAL;
    DECLARE
      environment   prcs$index,
      port          access$selector,
      destination   access$selector,
      carrier       access$selector;
  END
IF$surrogate$receive;


/*   function Send_to_processor(
/*       environment:      prcs_index;
/*       psor:             access_selector;
/*       message:          IPC_message;
/*       success:          boolean_acc)
/*     return boolean;              -- true if context-faulted */
```

```
IF$send$to$processor:
  PROCEDURE(environment, psor, message, success$ptr) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      psor           access$selector,
      message        IPC$message,
      success$ptr    POINTER;
  END
IF$send$to$processor;


/*   function Broadcast_to_processors(
/*       environment:       prcs_index;
/*       psor:              access_selector;
/*       message:           IPC_message;
/*       success:           boolean_acc)
/*     return boolean;                -- true if context-faulted */

IF$broadcast$to$processors:
  PROCEDURE(environment, psor, message, success$ptr) boolean EXTERNAL;
    DECLARE
      environment    prcs$index,
      psor           access$selector,
      message        IPC$message,
      success$ptr    POINTER;
  END
IF$broadcast$to$processors;


/*   function Send_to_physical_processor(
/*       message:           IPC_message;
/*       psor_lcom_adr:     GDP_physical_address;
/*       success:           boolean_acc)
/*     return boolean;                -- true if context-faulted */

IF$send$to$physical$processor:
  PROCEDURE(message, psor$lcom$adr$lower, psor$lcom$adr$upper,
          success$ptr) boolean EXTERNAL;
    DECLARE
      message               IPC$message,
      psor$lcom$adr$lower   short$ordinal,
      psor$lcom$adr$upper   short$ordinal,
      success$ptr           POINTER;
  END
IF$send$to$physical$processor;
```

```
/*   function Read_processor_status(
/*       environment:      prcs_index;
/*       destination:      processor_status_record_acc)
/*     return boolean;                -- true if context-faulted
/*
/*     -- NOTE: This is the logical mode operator only. */

IF$read$processor$status:
  PROCEDURE(environment, destination$ptr) boolean EXTERNAL;
    DECLARE
      environment      prcs$index,
      destination$ptr  POINTER; /* this is a pointer for the
                                   4-byte processor$status$record
                                   field where the result will be put */
  END
IF$read$processor$status;


/*   function Phy_read_processor_status(
/*       destination:      processor_status_record_acc)
/*     return boolean;                -- true if context-faulted
/*
/*     -- NOTE: This is the physical mode operator only. */

IF$phy$read$processor$status:
  PROCEDURE(destination$ptr) boolean EXTERNAL;
    DECLARE
      destination$ptr  POINTER; /* this is a pointer for the
                                   4-byte processor$status$record
                                   field where the result will be put */
  END
IF$phy$read$processor$status;


/*   function Indivisibly_add_short_ordinal(
/*       environment:           prcs_index;
/*       source:                access_selector;
/*       displacement:          short_ordinal;
/*       value:                 short_ordinal;
/*       original_value:        short_ordinal_acc)
/*     return boolean;             -- true if context-faulted */

IF$indivisibly$add$short$ordinal:
  PROCEDURE(environment, source, displacement, value,
            original$value$ptr) boolean EXTERNAL;
    DECLARE environment          prcs$index,
            source               access$selector,
            displacement         short$ordinal,
            value                short$ordinal,
            original$value$ptr   POINTER;
  END
IF$indivisibly$add$short$ordinal;
```

```
/*   function Indivisibly_insert_short_ordinal(
/*       environment:            prcs_index;
/*       source:                 access_selector;
/*       displacement:           short_ordinal;
/*       value:                  short_ordinal;
/*       mask:                   short_ordinal;
/*       original_value:         short_ordinal_acc)
/*     return boolean;              -- true if context-faulted */

IF$indivisibly$insert$short$ordinal:
  PROCEDURE(environment, source, displacement, value, mask,
      original$value$ptr) boolean EXTERNAL;
    DECLARE environment          prcs$index,
            source               access$selector,
            displacement         short$ordinal,
            value                short$ordinal,
            mask                 short$ordinal,
            original$value$ptr   POINTER;
  END
IF$indivisibly$insert$short$ordinal;


/*   function Dispatch(
/*       environment:            prcs_index;
/*       psor_os:                access_selector)
/*     return boolean;              -- true if context-faulted */

IF$dispatch:
  PROCEDURE(environment, psor$os) boolean EXTERNAL;
    DECLARE environment          prcs$index,
            psor$os              access$selector;
  END
IF$dispatch;
/*
/* end IP_Function_Interface; */
```

IP_TRANSFER_DRIVER PACKAGE

```
/* with IP_Basic_Definitions, AP_Executive_Calls;
/*
/* package IP_Transfer_Driver is
/*
/*   -- Function:
/*   --   Transfer_Driver provides a synchronous interface used by
/*   --   strategy routines to transfer a block of data between GDP
/*   --   memory and AP memory.
/*
/*   use IP_Basic_Definitions, AP_Executive_Calls;
/*
/*   function Transfer_to_AP(
/*       environment:       prcs_index;
/*         -- environment requesting the data transfer
/*       object:            access_selector;
/*         -- coordinates of object access descriptor in environment
/*       offset:            short_ordinal;
/*         -- byte offset into object of beginning of block
/*       length:            short_ordinal;
/*         -- byte length of block
/*       AP_array_ref:      pointer)
/*         -- AP memory block where data is to be put
/*     return boolean;   -- true if context-faulted or equivalent
/*
/*     -- Function:
/*     --   This procedure transfers a block of data from a data segment
/*     --   resident in i432 memory to the AP's memory. */

transfer$to$AP:
  procedure(environment, object, offset, length, AP$array$ref)
      boolean EXTERNAL;
    DECLARE environment     prcs$index,
            object          access$selector,
            offset          short$ordinal,
            length          short$ordinal,
            AP$array$ref    POINTER;
  END
transfer$to$AP;
```

```
/*   function Transfer_from_AP(
/*       environment:       prcs_index;
/*         -- environment requesting the transfer
/*       object:            access_selector;
/*         -- coordinates of object access descriptor in environment
/*       offset:            short_ordinal;
/*         -- byte offset into object of block beginning
/*       length:            short_ordinal;
/*         -- byte length of block
/*       AP_array_ref:      pointer)
/*         -- AP memory block where data is to be taken from
/*     return boolean;   -- true if context-faulted or equivalent
/*
/*     -- Function:
/*     --   This procedure tranfers a block of data from the AP's memory
/*     --   to a data segment resident in i432 memory. */

transfer$from$AP:
  procedure(environment, object, offset, length, AP$array$ref)
      boolean EXTERNAL;
    DECLARE environment     prcs$index,
            object          access$selector,
            offset          short$ordinal,
            length          short$ordinal,
            AP$array$ref    POINTER;
  END
transfer$from$AP;
/*
/* end IP_Transfer_Driver; */
```

## IP_FUNCTION_MANAGER PACKAGE

```
/* with IP_Basic_Definitions, AP_Executive_Calls;
/*
/* package IP_Function_Manager is
/*
/*   -- Function:
/*   --   IP_Function_Manager provides the lowest level interface to
/*   --   the IP function request facilities. Users of this package must
/*   --   be aware of the layout of parameters and results observed by
/*   --   the IP, and must be prepared to check the completion state
/*   --   of the operation and take appropriate action if that indicates
/*   --   conditions such as a fault or a blocked port operation.
/*
/*   use IP_Basic_Definitions, AP_Executive_Calls;
/*
/*
/*   -- A function_request record is a description of an operation and
/*   -- is passed to the Function Manager to get the operation
/*   -- executed.  The record contains all the parameters required by
/*   -- the IP and any additional ones needed by the Function Manager.
/*
/*   type function_request_record is
/*     record
/*       environment:      prcs_index;
/*         -- the IP process to use as an environment
/*       state:            function_state;
/*         -- the completion state of the operation
/*       opcode:           IP_operator;
/*         -- the IP operation code for the operation
/*       operands:         array(1 .. 7) of short_ordinal;
/*         -- the operands for the operator, usually a collection of
/*         -- access selectors and numeric values.  Defined by the IP.
/*       result:           array(1 .. 10) of short_ordinal;
/*         -- the results of the operation.  Defined by the IP.
/*       fault_result:     process_fault_reply;
/*         -- result returned by a process level fault
/*
/*         -- The following two are only valid for all variations of the
/*         -- Send and Receive port operations.
/*       delayed_response_port:    exchange;
/*         -- where to send message if operation completion was delayed.
/*       delayed_response_msg:     AP_message;
/*         -- the message to send
/*     end record; */
```

```
DECLARE function$request$record  LITERALLY 'STRUCTURE(
  environment  prcs$index,
  state        function$state,
  opcode       IP$operator,
  operands(7)  short$ordinal,
  result(10)   short$ordinal,
  fault$result process$fault$reply,
  delayed$response$port exchange,
  delayed$response$msg  AP$message)';
/*
/*   -- Accesses to the function request block are passed around rather
/*   -- than copies of the record.
/*
/*   type function_request is access function_request_record; */

DECLARE function$request  LITERALLY 'POINTER';

DECLARE              function$request$record$len               LITERALLY
'(function$request$facility$rec$len) + 10';
/*
/*   type faulting_message_record is
/*     record
/*       request:  function_request;  -- the function that got the fault
/*     end record; */

DECLARE faulting$message$record LITERALLY
         'STRUCTURE(request function$request)';

DECLARE faulting$message$record$len  LITERALLY '4';
/*
/*   type faulting_message is access faulting_message_record; */

DECLARE faulting$message  LITERALLY 'AP$message';


/*
/*   procedure Request_function(
/*       request:    function_request); -- operation description
/*
/*     -- Function:
/*     --   This procedure presents the request to the IP's function
/*     --   request facility and triggers its execution. Control is
/*     --   returned when the state of execution (contained in the
/*     --   parameter record) indicates that the IP has been released.
/*     --   The possible states are described in IP_Definition by the
/*     --   type function_completion_state.
/*     --
/*     --   Note that either logical or physical mode operators can be
/*     --   invoked via this interface. Physical mode can be selected
/*     --   by not having invoked the IP Controller initialization
/*     --   routine and by having an FFFF in the environment field of
/*     --   the specified request. */
```

```
request$function:
  PROCEDURE(request) EXTERNAL;
    DECLARE request  function$request;
  END
request$function;


/*   function Get_function_request(
/*       environment:      prcs_index)
/*         -- IP process whose function block is wanted
/*     return function_request;
/*         -- the function block associated with the IP process.
/*
/*     -- Function:
/*     --   Returns the function request block associated with the IP
/*     --   process.  This block is provided so that the function
/*     --   interface does not have to dynamically create and destroy
/*     --   the blocks.  No sychronization is provided for the blocks.
/* */

get$function$request:
  PROCEDURE(environment) function$request EXTERNAL;
    DECLARE environment   prcs$index;
  END
get$function$request;


/*   -- FAULTING INTERFACE
/*   --   The interface to operator faulting is to define an exchange
/*   --   associated with each IP process and each window.  If the
/*   --   reporting of a fault is delayed, such as when a process level
/*   --   fault occurs, then the operation is finished with a
/*   --   "fault_delayed" completion code.  When the process comes back
/*   --   from the fault handler, a message, consisting only of a
/*   --   pointer to the function request block that caused the fault,
/*   --   is sent to the exchange associated with the IP process.  This
/*   --   exchange is setable and retrievable.
/*
/*
/*   function Get_fault_exchange(
/*       environment:      prcs_index)
/*         -- IP process whose exchange is wanted
/*     return exchange;  -- the exchange associated with the IP process
/*
/*     -- Function:
/*     --   Returns the fault wait port currently associated with the
/*     --   IP process */

get$fault$exchange:
  PROCEDURE(environment) exchange EXTERNAL;
    DECLARE environment  prcs$index;
  END
get$fault$exchange;
```

```
/*   function Get_fault_message(
/*       environment:       prcs_index)
/*         -- IP process whose message is wanted
/*     return AP_message;  -- the message associated with the IP process
/*
/*     -- Function:
/*     --   Returns the fault wait message currently associated with
/*     --   the IP process */

get$fault$message:
  PROCEDURE(environment) AP$message EXTERNAL;
    DECLARE environment  prcs$index;
  END
get$fault$message;


/*   procedure Set_fault_exchange(
/*       environment:              prcs_index;
/*         -- the IP process whose exchange is to be set
/*       response_exchange:        exchange);
/*         -- the exchange to use
/*
/*     -- Function:
/*     --   Sets the exchange to use when the fault handler
/*     --   returns the environment after handling the fault. */

set$fault$exchange:
  PROCEDURE(environment, response$exchange) EXTERNAL;
    DECLARE environment         prcs$index,
            response$exchange   exchange;
  END
set$fault$exchange;


/*   -- ROUTINES CALLED BY THE INTERRUPT SERVICING TASK
/*   -- (FOR IP INTERRUPTS)
/*
/*
/*   procedure Handle_IPC;
/*
/*     -- Function:
/*     --   Most IPCs received by the IP are reflected to the AP for
/*     --   consideration.  This routine will case on the IPC code and
/*     --   perform the appropriate action.  Since the codes to be
/*     --   reflected are TBD, this routine is currently empty. */

handle$IPC:
  PROCEDURE EXTERNAL;
  END
handle$IPC;
```

```
/*   procedure Handle_alarm;
/*
/*     -- Function:
/*     --   An alarm signal was received from a source external to the
/*     --   AP system.  The cause of the alarm is system dependent and
/*     --   this routine may vary across configurations. */

handle$alarm:
  PROCEDURE EXTERNAL;
  END
handle$alarm;


/*   procedure Handle_reconfiguration;
/*
/*     -- Function:
/*     --   This should not occur until the 700 series systems. */

handle$reconfiguration:
  PROCEDURE EXTERNAL;
  END
handle$reconfiguration;


/*   procedure Handle_dispatching;
/*
/*     -- Function:
/*     --   The IP has indicated that a process has been received at its
/*     --   dispatching port. Its owner will be notified by sending the
/*     --   delay message or the fault message to the delay exchange or
/*     --   fault exchange. */

handle$dispatching:
  PROCEDURE EXTERNAL;
  END
handle$dispatching;


/*   procedure Bug_fix_for_dispatching;
/*
/*     -- Function:
/*     --   This procedure is necessary because of a bug in the 2.0 IP
/*     --   microcode which does not always clean up correctly after a
/*     --   receive-block on a message occurs. */

bug$fix$for$dispatching:
  PROCEDURE(environment) EXTERNAL;
    DECLARE environment  prcs$index;
  END
bug$fix$for$dispatching;
```

```
/*   procedure Frmgr_init;
/*
/*     -- Function:
/*     --   This procedure prepares the module for future dispatching.
/* */

frmgr$init:
  PROCEDURE EXTERNAL;
  END
frmgr$init;
/*
/* end IP_Function_Manager; */
```

IP_WINDOW_MANAGER PACKAGE

```
/* with IP_Basic_Definitions, AP_Executive_Calls, IP_Function_Interface;
/*
/* package IP_Window_Manager is
/*
/*   -- Function:
/*   --   IP_Window_Manager provides a synchronous interface for
/*   --   the allocation/deallocation of both the IP windows and the
/*   --   AP memory range that is mapped via the windows.
/*
/*   use IP_Basic_Definitions, AP_Executive_Calls,
/*       IP_Function_Interface;
/*
/*   type transfer_direction is (i432_to_AP, AP_to_i432); */

DECLARE transfer$direction   LITERALLY 'WORD';
DECLARE i432$to$AP           LITERALLY '0';
DECLARE AP$to$i432           LITERALLY '1';
/*
/*   -- An "io_segment" records information about a specific allocation
/*   -- of an IP window and an associated range of AP memory space.
/*   -- The information it contains comes in two flavors:  a base
/*   -- address that the holder must see in order to use the
/*   -- allocation, and other information that the holder should never
/*   -- alter (which is also used as a token for deallocation purposes).
/*
/*   type window_pointer is new pointer; */

DECLARE window$pointer   LITERALLY 'POINTER';
/*
/*   type internal_window_description is
/*     record
/*       window_number:         alterable_window_index;
/*         -- which window to manipulate
/*       description_is_valid:  boolean := false;
/*     end record; */

DECLARE internal$window$description   LITERALLY '
    window$number          alterable$window$index,
    description$is$valid   boolean';
```

```
/*   type io_segment_representation is
/*     record
/*       window_block:     window_pointer;
/*         -- global pointer to window
/*       description:      internal_window_description;
/*         -- hidden description to allow
/*         -- deallocation checking
/*     end record; */

DECLARE io$segment$representation  LITERALLY 'STRUCTURE(
    window$block   window$pointer,
    internal$window$description)';
/*
/*   type io_segment is access io_segment_representation; */

DECLARE  io$segment   LITERALLY 'POINTER';


/*
/*     -- This array tracks window faults. Users should check the
/*     -- element corresponding to any window used before closing it.
/*
/*   window_faulted:  array (window_index) of boolean :=
/*                      (others => false); */

DECLARE window$faulted(5) boolean EXTERNAL;


/*   function Get_window(
/*       xfer_mode:          transfer_direction;
/*         -- direction to or from 432
/*       environment:        prcs_index;
/*         -- IP process index
/*       object:             access_selector;
/*         -- object to be mapped
/*       offset:             short_ordinal;
/*         -- displacement into object
/*       length:             short_ordinal;
/*         -- size of area to map
/*       io_seg:    in out io_segment)
/*         -- desription record of the allocation
/*     return boolean;              -- true if context-faulted
/*
/*     -- Function:
/*     --   Allocates and opens an IP window into 432 memory.
/*     --   The io_segment parameter contains a semi-private description
/*     --   of the allocation, which can be used by the caller to access
/*     --   the allocation and also return the allocation when finished.
/*     --
/*     --  NOTE: The caller must allocate the block for the io_segment,
/*     --        and pass in a reference to it.
/*     -- */
```

```
get$window:
  PROCEDURE(xfer$mode, environment, object, offset, length, io$seg)
     boolean EXTERNAL;
    DECLARE xfer$mode       transfer$direction,
            environment     prcs$index,
            object          access$selector,
            offset          short$ordinal,
            length          short$ordinal,
            io$seg          io$segment;
  END
get$window;


/*   function Return_window(
/*       environment:      prcs_index;  -- IP process index
/*       mapping:          io_segment)  -- AP memory range currently
/*                                      -- mapped via a window
/*     return boolean;                  -- true if context-faulted
/*
/*     -- Function:
/*     --   Closes and releases the window indicated by the parameter
/*     --   and then marks the io_segment as being released.
/*     -- */

return$window:
  PROCEDURE(environment, mapping) boolean EXTERNAL;
    DECLARE environment   prcs$index,
            mapping       io$segment;
  END
return$window;


/*   procedure Handle_window_fault(
/*       window_number:    window_index);  -- window having the fault
/*
/*     -- Function:
/*     --   The corresponding element in window_faulted is set, and the
/*     --   subsystem is brought down if the fault was for window 4.
/*     --
/*     --   This procedure is called by the interrupt routine when it
/*     --   is notified of a window fault. */

handle$window$fault:
  PROCEDURE(window$number) EXTERNAL;
    DECLARE window$number   window$index;
  END
handle$window$fault;

/* end IP_Window_Manager; */

window$init:
  PROCEDURE EXTERNAL;
  END
window$init;
```

INITLO PACKAGE BODY

```
/**********

    Title:  INITLZ - initialization task for the AP I/O system.
                ** NO TERMINAL VERSION **

    Logic:  This module is user-modifiable. This initial task
            calls an IP Controller initialization routine, allocates
            and gives to the free space manager space for all
            allocations required, and can optionally call
            any other user-defined routine.

**********/

initlz: DO;

$INCLUDE(:f1:basdef.inc)
$INCLUDE(:f1:apexec.inc)
$INCLUDE(:f1:apini0.ext)

  /* The following array is given to the free space manager.
     The amount given is more than sufficient for debugging the
     5-terminal version with 30 IP processes in the system.
  */
$IF NOT megabyte
  DECLARE fs$bytes     LITERALLY '500';
$ELSE
  DECLARE fs$bytes    LITERALLY '25000';
$ENDIF

  DECLARE fs$array(fs$bytes)  BYTE;

  /* When true, AP system errors will be displayed at the 86/12A serial
     port terminal, as well as at the standard sys_error reporting
     mechanism. */
  DECLARE display_sys_errors boolean DATA(true);

  start$init:
    PROCEDURE PUBLIC;

$IF not megabyte
      fs$array(2) = LOW(fs$bytes);
      fs$array(3) = HIGH(fs$bytes);
$ELSE
      fs$array(4) = LOW(fs$bytes);
      fs$array(5) = HIGH(fs$bytes);
$ENDIF
      CALL AP$return$space(@ fs$array);
      /* set-up IP Controller to handle 0 processes */
      CALL initialize$IP$controller(0, display_sys_errors);

      /* This is where users should insert their own code. */

      CALL AP$suspend$self;

    END
  start$init;

END initlz;
```

INITL1 PACKAGE BODY

```
/**********

    Title:  INITLZ - initialization task for the AP I/O system.
                ** 1 TERMINAL VERSION **

    Logic:  This module is user-modifiable. This initial task
            calls an IP Controller initialization routine, allocates
            and gives to the free space manager space for all
            allocations required, calls a routine to startup the
            86/12 serial port support, and can optionally call
            any other user-defined routine.

**********/

initlz: DO;

$INCLUDE(:f1:basdef.inc)
$INCLUDE(:f1:apexec.inc)
$INCLUDE(:f1:apini1.ext)

  /* The following array is given to the free space manager.
      The amount given is more than sufficient for debugging the
      5-terminal version with 30 IP processes in the system.
  */
$IF NOT megabyte
  DECLARE fs$bytes     LITERALLY '5500';
$ELSE
  DECLARE fs$bytes    LITERALLY '25000';
$ENDIF

  DECLARE fs$array(fs$bytes)  BYTE;

  /* legal baud rates are represented as follows :
         0 =   150
         1 =   300
         2 =   600
         3 =  1200
         4 =  2400
         5 =  4800
         6 =  9600
         7 = 19200                    */

  /* The initial baud rates. */
  DECLARE init$8612$terminal$speed      WORD DATA(4);
  /* The initial task priorities and number of characters per terminal
     line */
  DECLARE terminal$support$task$priorities WORD DATA(150),
          number$of$characters$in$line     WORD DATA(136);

  /* The initial standard terminal IP process index assignments */
  DECLARE std$term$output$first$IP$env     WORD DATA(1),
          std$term$output$last$IP$env      WORD DATA(2),
          std$term$input$first$IP$env      WORD DATA(3),
          std$term$input$last$IP$env       WORD DATA(4);

  /* When true, AP system errors will be displayed at the 86/12A serial
     port terminal, as well as at the standard sys_error reporting
     mechanism. */
  DECLARE display_sys_errors boolean DATA(true);

  start$init:
    PROCEDURE PUBLIC;

$IF not megabyte
      fs$array(2) = LOW(fs$bytes);
      fs$array(3) = HIGH(fs$bytes);
$ELSE
      fs$array(4) = LOW(fs$bytes);
      fs$array(5) = HIGH(fs$bytes);
$ENDIF
      CALL AP$return$space(@ fs$array);

      /* set-up IP Controller to handle 4 processes */
      CALL initialize$IP$controller(4, display_sys_errors);

      CALL initialize$8612$terminal(init$8612$terminal$speed,
                                    terminal$support$task$priorities,
                                    number$of$characters$in$line,
                                    std$term$output$first$IP$env,
                                    std$term$output$last$IP$env,
                                    std$term$input$first$IP$env,
                                    std$term$input$last$IP$env);

      /* This is where users should insert their own code. */

      CALL AP$suspend$self;

    END
  start$init;

END initlz;
```

## INITL5 PACKAGE BODY

```
/**********

    Title:  INITLZ - initialization task for the AP I/O system.

    Logic:  This module is user-modifiable. This initial task
            calls an IP Controller initialization routine, allocates
            and gives to the free space manager space for all
            allocations required, calls a routine to startup the
            86/12 serial port support, calls a routine to startup
            the 534 serial ports support, and can optionally call
            any other user-defined routine.

**********/

initlz: DO;

$INCLUDE(:f1:basdef.inc)
$INCLUDE(:f1:apexec.inc)
$INCLUDE(:f1:apini5.ext)

  /* The following array is given to the free space manager.
      The amount given is more than sufficient for debugging the
      5-terminal version with 30 IP processes in the system.
  */
$IF NOT megabyte
  DECLARE fs$bytes     LITERALLY '25000';
$ELSE
  DECLARE fs$bytes    LITERALLY '25000';
$ENDIF

  DECLARE fs$array(fs$bytes)  BYTE;

  /* legal baud rates are represented as follows :
        0 =   150
        1 =   300
        2 =   600
        3 =  1200
        4 =  2400
        5 =  4800
        6 =  9600
        7 = 19200                  */

  /* The initial baud rates. */
  DECLARE init$8612$terminal$speed      WORD DATA(4);
  DECLARE init$534$terminal$speeds(4)   WORD DATA(4, 4, 4, 4);
  /* The initial task priorities and number of characters per terminal
     line */
  DECLARE terminal$support$task$priorities WORD DATA(150),
          number$of$characters$in$line     WORD DATA(136);

  /* The initial standard terminal IP process index assignments */
  DECLARE std$term$output$first$IP$env     WORD DATA(1),
          std$term$output$last$IP$env      WORD DATA(2),
          std$term$input$first$IP$env      WORD DATA(3),
          std$term$input$last$IP$env       WORD DATA(4);

  /* The initial 534 terminal IP process index assignments */
  DECLARE term$534$output$first$IP$env(4)  WORD DATA(5,9,13,17),
          term$534$output$last$IP$env(4)   WORD DATA(6,10,14,18),
          term$534$input$first$IP$env(4)   WORD DATA(7,11,15,19),
          term$534$input$last$IP$env(4)    WORD DATA(8,12,16,20);

  /* When true, AP system errors will be displayed at the 86/12A serial
     port terminal, as well as at the standard sys_error reporting
     mechanism. */
  DECLARE display_sys_errors boolean DATA(true);

  start$init:
    PROCEDURE PUBLIC;

$IF not megabyte
      fs$array(2) = LOW(fs$bytes);
      fs$array(3) = HIGH(fs$bytes);
$ELSE
      fs$array(4) = LOW(fs$bytes);
      fs$array(5) = HIGH(fs$bytes);
$ENDIF
      CALL AP$return$space(@ fs$array);

      /* set-up IP Controller to handle 20 processes */
      CALL initialize$IP$controller(20, display_sys_errors);

      CALL initialize$8612$terminal(init$8612$terminal$speed,
                                    terminal$support$task$priorities,
                                    number$of$characters$in$line,
                                    std$term$output$first$IP$env,
                                    std$term$output$last$IP$env,
                                    std$term$input$first$IP$env,
                                    std$term$input$last$IP$env);
      CALL initialize$534$terminals(@ init$534$terminal$speeds,
                                    terminal$support$task$priorities,
                                    number$of$characters$in$line,
                                    @ term$534$output$first$IP$env,
                                    @ term$534$output$last$IP$env,
                                    @ term$534$input$first$IP$env,
                                    @ term$534$input$last$IP$env);

      /* This is where users should insert their own code. */

      CALL AP$suspend$self;

    END
  start$init;

END initlz;
```

APINIO PACKAGE

```
/**********

    Title:  APINIO - AP initialization utility for 0 terminals.

    Logic:  This routine is made available to the AP initialization
            routine so that it can bring up the IP Controller.

**********/


/**********

    INITIALIZE IP CONTROLLER: This routine completely initializes the
        IP Controller.
          NOTE: Only the physical mode operators in the IP Controller
          can be used before this routine is called. This routine waits
          for the IP to go into logical mode as part of this routine.
          After this routine is called, only the logical mode operators
          in the IP Controller should be used.

**********/

initialize$IP$controller:
  PROCEDURE(last$IP$process, display_sys_errors) EXTERNAL;
    DECLARE last$IP$process     prcs$index,
              /* index of last IP process */
            display_sys_errors  boolean;
              /* switch which allows AP system errors to be
                 displayed at the 86/12A serial port terminal */
  END
initialize$IP$controller;
```

APINI1 PACKAGE

```
/**********

    Title:  APINI1 - AP initialization utilities for 1 terminal.

    Logic:  These routines are made available to the AP initialization
            routine so it can bring up the IP Controller
            and the desired I/O support software.

**********/


/**********

    INITIALIZE IP CONTROLLER: This routine completely initializes the
        IP Controller.
          NOTE: Only the physical mode operators in the IP Controller
          can be used before this routine is called. This routine waits
          for the IP to go into logical mode as part of this routine.
          After this routine is called, only the logical mode operators
          in the IP Controller should be used.

**********/

initialize$IP$controller:
  PROCEDURE(last$IP$process, display_sys_errors) EXTERNAL;
    DECLARE last$IP$process     prcs$index,
              /* index of last IP process */
            display_sys_errors  boolean;
              /* switch which allows AP system errors to be
                 displayed at the 86/12A serial port terminal */
  END
initialize$IP$controller;


/**********

    INITIALIZE 8612 TERMINAL: This routine configures in the software
        support for a terminal on the serial I/O port of the 86/12
        board. The actual terminal handler tasks are configured in by
        the RMX/88 Interactive Configuration Utility - not by this
        routine. This routine is merely responsible for starting the
        strategy and monitor tasks.

**********/
```

```
initialize$8612$terminal:
    PROCEDURE(init$baud$rate, terminal$support$task$priorities,
              number$of$characters$in$line,
              std$terminal$output$first$IP$env,
              std$terminal$output$last$IP$env,
              std$terminal$input$first$IP$env,
              std$terminal$input$last$IP$env) EXTERNAL;
      DECLARE
         init$baud$rate                     WORD,
           /* initial baud rate for standard terminal */
         terminal$support$task$priorities   WORD,
           /* task priority for terminal support */
         number$of$characters$in$line       WORD,
           /* number of characters in a terminal line */
         std$terminal$output$first$IP$env   WORD,
           /* first IP process assignment for output */
         std$terminal$output$last$IP$env    WORD,
           /* last IP process assignment for output */
         std$terminal$input$first$IP$env    WORD,
           /* first IP process assignment for input */
         std$terminal$input$last$IP$env     WORD;
           /* last IP process assignment for input */
  END
initialize$8612$terminal;
```

## APINI5 PACKAGE

```
/**********

    Title:  APINI5 - AP initialization utilities.

    Logic:  These routines are made available to the AP initialization
            routine so it can bring up the IP Controller
            and the desired I/O support software.

**********/


/**********

    INITIALIZE IP CONTROLLER: This routine completely initializes the
                              IP Controller.
          NOTE: Only the physical mode operators in the IP Controller
          can be used before this routine is called. This routine waits
          for the IP to go into logical mode as part of this routine.
          After this routine is called, only the logical mode operators
          in the IP Controller should be used.

**********/

initialize$IP$controller:
  PROCEDURE(last$IP$process, display_sys_errors) EXTERNAL;
    DECLARE last$IP$process     prcs$index,
              /* index of last IP process */
            display_sys_errors  boolean;
              /* switch which allows AP system errors to be
                 displayed at the 86/12A serial port terminal */
  END
initialize$IP$controller;


/**********

    INITIALIZE 8612 TERMINAL: This routine configures in the software
        support for a terminal on the serial I/O port of the 86/12
        board. The actual terminal handler tasks are configured in by
        the RMX/88 Interactive Configuration Utility - not by this
        routine. This routine is merely responsible for starting the
        strategy and monitor tasks.

**********/
```

```
initialize$8612$terminal:
    PROCEDURE(init$baud$rate, terminal$support$task$priorities,
              number$of$characters$in$line,
              std$terminal$output$first$IP$env,
              std$terminal$output$last$IP$env,
              std$terminal$input$first$IP$env,
              std$terminal$input$last$IP$env) EXTERNAL;
      DECLARE
         init$baud$rate                      WORD,
           /* initial baud rate for standard terminal */
         terminal$support$task$priorities   WORD,
           /* task priority for terminal support */
         number$of$characters$in$line       WORD,
           /* number of characters in a terminal line */
         std$terminal$output$first$IP$env   WORD,
           /* first IP process assignment for output */
         std$terminal$output$last$IP$env    WORD,
           /* last IP process assignment for output */
         std$terminal$input$first$IP$env    WORD,
           /* first IP process assignment for input */
         std$terminal$input$last$IP$env     WORD;
           /* last IP process assignment for input */
  END
initialize$8612$terminal;


/**********

    INITIALIZE 534 TERMINALS: This routine configures in the software
        support for 4 terminals on the serial ports of the 534 board.
        This routine starts up the corresponding terminal handler tasks
        as well as the strategy and monitor tasks.

**********/

initialize$534$terminals:
    PROCEDURE(init$baud$rate$ptr, terminal$support$task$priorities,
              number$of$characters$in$line,
              term$534$output$first$IP$env$ptr,
              term$534$output$last$IP$env$ptr,
              term$534$input$first$IP$env$ptr,
              term$534$input$last$IP$env$ptr) EXTERNAL;
      DECLARE
         init$baud$rate$ptr                 POINTER,
           /* pointer to an array of initial baud
              rates for the 534 terminals */
         terminal$support$task$priorities   WORD,
           /* task priority for terminal support */
         number$of$characters$in$line       WORD,
           /* number of characters in a terminal line */
         term$534$output$first$IP$env$ptr   POINTER,
           /* pointer to an array of first IP process assignments
              for output on 534 terminals */
         term$534$output$last$IP$env$ptr    POINTER,
           /* pointer to an array of last IP process assignments
              for output on 534 terminals */
         term$534$input$first$IP$env$ptr    POINTER,
           /* pointer to an array of first IP process assignments
              for input on 534 terminals */
         term$534$input$last$IP$env$ptr     POINTER;
           /* pointer to an array of last IP process assignments
              for input on 534 terminals */
    END
initialize$534$terminals;
```

IP_PROCESSES PACKAGE BODY

```
with Untyped_Ports, iMAX_Definitions, IP_Management, IO_Devices,
     Actual_Terminal_Sinks,
     Actual_Terminal_Sources, Terminal_Interfaces,
     Synchronous_IO_Interfaces,
     Unchecked_conversion, Asynchronous_IO_Interface, Memory_Controller,
     Debug_Sink, Debug_Source;
package body IP_Processes is
  --
  -- Specification file: IPPRCS.MBS
  --
  -- Logic:
  --

  use IO_Devices, Synchronous_IO_Interfaces;



   -------------------------
  --                       --
  -- NUMBER OF CONNECTIONS --
  --                       --
  -------------------------


  --
  -- IP connections (two ports for each).
  --
     number_of_connections: constant := 1;   -- (1 terminal)
  --
  -------------------------


  -- Definitions for controlling starting IPs and creating
  -- synchronous interfaces.

  type control_switches is (start_ip, num_connections);

  type c_array_rep_val is array (control_switches) of short_ordinal;

  type c_array_rep is access c_array_rep_val;

  -- now define a constant array to determine whether to
  -- start IPs or not (0 =] don't start the IP) and the
  -- number of connections supported.

  control_array:
    constant c_array_rep := new c_array_rep_val(;
      start_ip =] 0,
      num_connections =] number_of_connections);


   -- Define a record for IP process globals access segments.
```

```
type IP_process_globals_rep_val is
  record
    request_port: iMAX_Definitions.port;
  end record;

type IP_process_globals_rep is access IP_process_globals_rep_val;


-- IP process configuration parameters

IP_processor_id:       constant := 5;
```

```
number_of_msg_slots:   constant := 10;  -- message slots in I/O ports
number_of_processes:   constant := 2;   -- processes per in or out
                                        -- port
number_of_work_ADs:    constant := 10;  -- context AS work space
                                        -- (slot 9 is assumed to
                                        -- be a null by AP software)
number_of_work_bytes:  constant := 0;

debug_source_AD: Source retypes Debug_Source;
debug_sink_AD:   Sink retypes Debug_Sink;


procedure Initialize
  --
is
  prcs_no: short_ordinal := 0;
  tport:   iMAX_Definitions.port;
  tname:   iMAX_Definitions.print_name;
  tpglob:  IP_process_globals_rep;

  --** Due to a bug in IP faulting, the ports (and messages) used in
  --** communication with the IP need to be in frozen memory.
  SRO:     iMAX_Definitions.storage_resource :=
             Memory_Controller.frozen_global_heap_SRO;
begin
  -- Create IO devices from the debug source and sink.
  IO_Devices.debug_source := debug_source_AD;
  IO_Devices.debug_sink := debug_sink_AD;

  -- Create the console from the debugger sink
  IO_Devices.console := debug_sink_AD;
```

```
    for i in interface_range range 1 .. number_of_connections loop

      -- create output side of connection
      tport := Untyped_Ports.Create_port(number_of_msg_slots,
                                         Untyped_Ports.FIFO,
                                         SRO);
      tname := "terminal 000   ";
      tname(12) := character'val(character'pos('0') + integer(i));
      tpglob := new IP_process_globals_rep_val(request_port => tport);
      terminals(i).tsink :=
        Actual_Terminal_Sinks.Create_terminal_sink(tport, tname);
      terminals(i).sink := Sink at
        terminals(i).tsink.Interface_Description;

      for j in 1 .. number_of_processes loop
        prcs_no := prcs_no + 1;
        IP_Management.Create_IP_process(IP_processor_id,
                                        prcs_no,
                                        number_of_work_ADs,
                                        number_of_work_bytes,
                                        any_access(tpglob));
      end loop;

      -- create input side of connection
      tport := Untyped_Ports.Create_port(number_of_msg_slots,
                                         Untyped_Ports.FIFO,
                                         SRO);
      tname := "terminal 000   ";
      tname(12) := character'val(character'pos('0') + integer(i));
      tpglob := new IP_process_globals_rep_val(request_port => tport);
      terminals(i).tsrc :=
        Actual_Terminal_Sources.Create_terminal_source(tport, tname);
      terminals(i).source := Source at
        terminals(i).tsrc.Interface_Description;

      for j in 1 .. number_of_processes loop
        prcs_no := prcs_no + 1;
        IP_Management.Create_IP_process(IP_processor_id,
                                        prcs_no,
                                        number_of_work_ADs,
                                        number_of_work_bytes,
                                        any_access(tpglob));
      end loop;
    end loop;

    -- Start the IP

    if control_array(start_ip) /= 0 then
      IP_Management.Start_IP(IP_processor_id);
    end if;

  end Initialize;
end IP_Processes;
```

ACTUAL_TERMINAL_SOURCES PACKAGE

```
with Terminal_Interfaces, iMAX_Definitions;
package Actual_Terminal_Sources is
  --
  -- Function:
  --   This package defines the actual domains for terminal
  --   source support.
  --

  function Create_terminal_source(
      req_port_parm:  iMAX_Definitions.port;
        -- request port for connection in new source
      conn_name_parm: iMAX_Definitions.print_name)
        -- name for connection in new source
    return Terminal_Interfaces.Terminal_Source;

    -- Function:
    --   This routine creates and returns a terminal source package.

end Actual_Terminal_Sources;
```

ACTUAL_TERMINAL_SINKS PACKAGE

```
with Untyped_Ports, Terminal_Interfaces, iMAX_Definitions;
package Actual_Terminal_Sinks is

  -- Function:
  --   This package defines the actual domains for terminal
  --   sink support.


  function Create_terminal_sink(
      req_port_parm:  iMAX_Definitions.port;
        -- request port for connection in new sink
      conn_name_parm: iMAX_Definitions.print_name)
        -- name for connection in new sink
    return Terminal_Interfaces.Terminal_Sink;

    -- Function:
    --   This routine creates and returns a terminal sink package.

end Actual_Terminal_Sinks;
```

IP_MANAGEMENT PACKAGE

```
with Descriptor_Definitions, iMAX_Definitions;
package IP_Management is

  -- Function:
  --   This package defines the operations necessary to create and
  --   install processes in IP processors, and initialize and start IP
  --   processors.


  procedure Create_IP_process(
      psor_num:            short_ordinal;
        -- ID of processor into which the new process is to be installed
      prcs_id:             short_ordinal;
        -- process ID (and offset in processor access object) of new
        -- process
      context_work_ADs:    short_ordinal;
        -- number of access descriptors in context access work area
      context_work_bytes: short_ordinal;
        -- number of bytes in context data work area
      pglob_as:            any_access);
        -- process globals access segment for IP process

    -- Function:
    --   This routine creates an IP process object with context access
    --   and data work areas of the requested length.  The access
    --   segment supplied as pglob_as is used as the process globals
    --   access segment for the IP process.

  procedure Start_IP(
      psor_num: short_ordinal);  -- processor ID of the IP to be started

    -- Function:
    --   This routine starts an Interface Processor.  It should be
    --   called after the IP processes have been created and
    --   initialized.

end IP_Management;
```

# CHAPTER CON
# CONFIGURATION

This chapter describes how the iMAX user specifies the system configuration on which the user's program, linked with iMAX, will run. The iMAX user can control these aspects of the configuration:

- number and processor identification of General Data Processors (GDPs) and Interface Processors (IPs) in the configuration

- static user processes, defined at compile-time and started at system initialization

- I/O device interfaces present in the configuration

Configuration of I/O device interfaces is described in Chapter IOI, Input/Output Implementation. This chapter (CON) only describes configuration of processors and of user static processes.

Note that minimal iMAX does not support Interface Processors or extensible I/O. Thus in minimal iMAX, configuration information consists only of a description of the GDPs and user static processes in the system.

The amount and type of memory present in the system is described via directives to the LINK-432 linker and not through iMAX configuration control. LINK-432 is described in the manual Intel 432 Cross Development System VAX/VMS Host User's Guide.

This chapter does not contain hardware configuration requirements (described in Appendix HDW, Hardware Configuration).

All configuration information resides in package bodies which the user may alter. The package specification, environment file, and Ada source file for each configuration package body are provided as part of iMAX. The user can modify, recompile, and replace the default body for any configuration package using the 432 Ada Compiler System and the LINK-432 linker.

## PROCESSORS

Each functioning processor in a system requires a processor object, and iMAX initialization starts all GDPs for which it finds a processor object. Because there is a relatively small space penalty for configuring more processors than are actually present, a system is can be configured for the maximum number of processors that can be physically installed. Processor boards can then be added to or removed from the system as desired without software reconfiguration.

The distributed version of iMAX is configured for two GDPs and two IPs. This configuration can be changed by modifying and recompiling the body of the package Processors. Note that to configure a system with more than six processors, the user must modify the constant number of psors in the Processors package body. Processors are configured by instantiating the generic package GDP Def with the processor identification. Each processor's identification number must be unique and be in the range 1..255.

For 432/600 systems there is a correspondence between processor identification number and the bus slot in which the processor card is installed (see Table CON-1).

Table CON-1. Processor ID Number to Bus Slot Mapping for 432/600 Systems

| 6-Slot System Bus Backplane | | 12-Slot System Bus Backplane | | 18-Slot System Bus Backplane | |
|---|---|---|---|---|---|
| SLOT | ID | SLOT | ID | SLOT | ID |
| 4 | 1 | 8 | 1 | 12 | 1 |
| 5 | 2 | 9 | 2 | 13 | 2 |
| 6 | 3 | 10 | 3 | 14 | 3 |
| | | 11 | 4 | 15 | 4 |
| | | 12 | 5 | 16 | 5 |
| | | | | 17 | 6 |

The processor slots in 432/600 systems can be used for either GDPs or IPs. There must be at least one GDP and one IP present. Thus the maximum number of GDPs is four in a 12-slot 432/600 system with one IP.

The default processors body at the end of this chapter illustrates how a user would configure a two-GDP and two-IP system for a 12-slot 432/670 system.

**WARNING**

Note that the user should not change the name of the processor list nor the code for procedure Initialize.

## STATIC PROCESSES

This section describes iMAX support for static user processes. iMAX allows any number of static user processes to be defined at compile time and started at system initialization (described in Chapter INI, Initialization). iMAX does not provide any control operations (start, stop, destroy) on static processes. Users requiring these capabilities should use iMAX dynamic processes, described in Chapter BPM, Basic Process Management. The process operations of iMAX Basic Process Management cannot be applied to static processes.

Static processes are most important in minimal iMAX systems. Minimal iMAX does not support dynamic user processes and all user processes must be defined at compile time as static processes.

### STATIC PROCESS OPERATIONS

The only operations available on static processes are the iMAX_Definitions.Idle procedure, which suspends the calling process for a given time period (described in Chapter DEF, Basic Definitions) and the port operations (described in Chapter COM, Interprocess Communication).

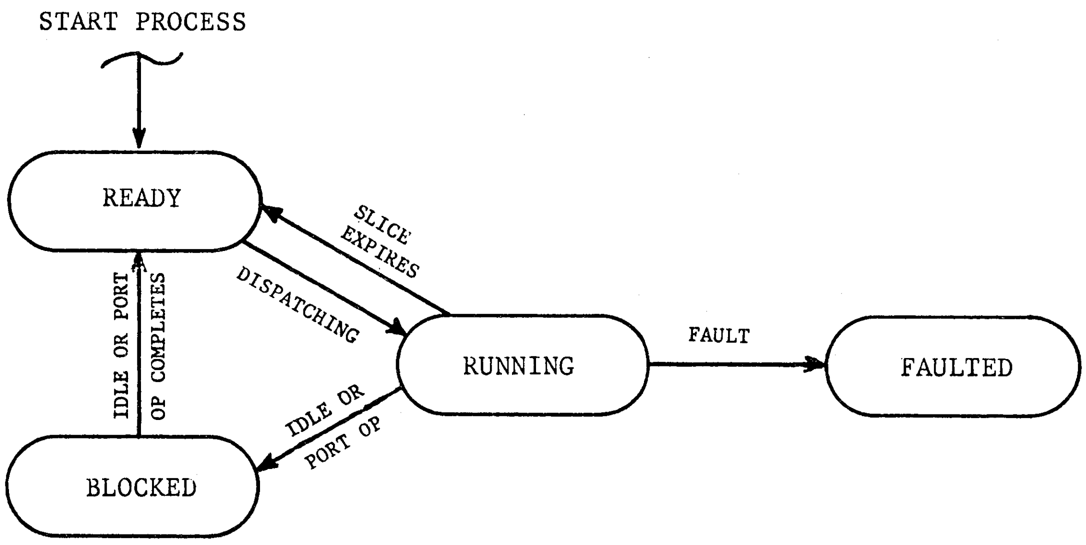


Figure CON-1. Static Process State Transitions

## STATIC PROCESS STATE TRANSITIONS

The state transitions for static processes are illustrated in Figure CON-1:

- Static processes are started in the ready state, queued at the dispatching port waiting for a processor.
- A running static process can be blocked because it invokes Idle or a blocking port operation. When the Idle or port operation completes, the process becomes ready again.
- A running static process may also give up a processor because its time slice expires. Such a process is rescheduled at the dispatching port among other ready processes.
- If a static process faults for any reason, the DEBUG-432 debugger reports the fault on the debugger console.

If a static process returns from its initial procedure, a system error results. System errors are described in Apendix FLT, Fault-Handling.

Note that static process states are different than the dynamic process states described in Chapter BPM, Basic Process Management.

## CREATING AND STARTING STATIC PROCESSES

Users can create and start static processes by writing a body for the package User_Processes, calling on definitions in the package Process_Definitions.

The User_Processes package specification contains only one entry, a parameterless procedure called Initialize. The user-supplied body (see example at end of this chapter) must instantiate all user processes and provide the body of Initialize to start them.

User processes are instantiated as occurences of the generic package Process_Instance, defined within Process_Definitions. The declarer supplies these parameters:

| | |
|---|---|
| Process_Startup | the initial procedure for the process |
| process_name | optional print_name for the process (defaults to all spaces) |
| SRO_size | optional size of process stack SRO in bytes (if defaulted or zero, iMAX selects a size) |
| num_OT_entries | optional number of entries in process object table (if defaulted or zero, iMAX selects a size) |

| | |
|---|---|
| frozen | optionally, whether the static process should be allocated in frozen memory (described in Chapter STO, Storage Management) (if defaulted, the static process is allocated in normal memory). |
| service_period | optional process scheduling parameter |
| deadline | optional process scheduling parameter |
| priority | optional process scheduling parameter |

Each instance of Process_Instance makes visible two fields: process id, the unique short ordinal identification of the created process, and the procedure Complete process initialization, which must be called by the user-supplied body of User_Processes.Initialize to start the process. The process_id field is not defined until Complete_process_initialization returns.

There is no limit on the number of user static processes. However, the user must direct the LINK-432 linker to reserve a sufficient number of object table directory (OTD) entries for the user processes. Each user process has its own object table, requiring a single OTD entry. The user should request (5+x) OTD entries, where x is the number of user static processes. LINK-432 linker directives are described in the Intel 432 Cross Development System VAX/VMS Host User's Guide.

## STATIC PROCESS SCHEDULING

Static processes contend only with each other for scheduling in minimal iMAX systems -- there are no run-time processes required to support minimal iMAX after initialization completes. In full iMAX systems, static processes contend both with each other and with dynamic user processes and system processes for scheduling.

The default scheduling parameters for static processes are given in Chapter DEF, Basic Definitions. Chapter DEF also explains the meaning of the scheduling parameters. Appendix PRS, Process Scheduling Information, gives guidance for choosing process scheduling parameter values. The iAPX 432 General Data Processor Architecture Reference Manual also describes process scheduling.

## PROCESSORS PACKAGE BODY

```
with IP_Initialization, iMAX_Definitions, Processor_Initialization,
     Unchecked_Conversion;
package body Processors is

  -- Specification file: PSORS.MSS

  -- Logic:
  --   Define the processors and the processor list.  The Initialize
  --   procedure is called during system initialization to make the
  --   processor list available to the system.

  --   This package is included in iMAX to allow reconfiguration of
  --   processors.  More processors may be added, or the mix of
  --   GDPs and IPs may be changed as desired.

  --   The variable name "processor_list" is used by LINK-432 to find
  --   the processor list.  THIS NAME CANNOT BE CHANGED.

  use iMAX_Definitions, Processor_Initialization, IP_Initialization;


  number_of_psors: constant := 6;
      -- 432/670 systems are limited to 6 processors

  subtype actual_psor_range is short_ordinal range 1 .. number_of_psors;

  type actual_psor_list_rep_val is array (actual_psor_range) of
                                     iMAX_Definitions.processor;
  type actual_psor_list_rep is access actual_psor_list_rep_val;

  -- Invoke the processor generics to create instances of GDP and IP
  -- processors as required.

  package psor1 is new GDP_Def(psor_num => 1);
  package psor2 is new GDP_Def(psor_num => 2);


  -- two IP processors, one for I/O, one for the debugger

  package psor4 is new IP_Def(psor_num   => 4,
                              prcs_count => 1,
                              psor_name  => "d_e_b_u_g_g_e_r");
  package psor5 is new IP_Def(psor_num   => 5,
                              prcs_count => 10,
                              psor_name  => " AP System One ");
  -- get a handle of the proper type on the processor objects

  processor1: processor retypes psor1.psor;
  processor2: processor retypes psor2.psor;
  processor3: constant processor := null;  -- default version has only
                                           -- 2 GDP's.
  processor4: processor retypes psor4.psor;
  processor5: processor retypes psor5.psor;


  -- processor 6 not currently provided for

  processor6: constant processor := null;


  processor_list: constant actual_psor_list_rep := new
                             actual_psor_list_rep_val(
                                                     1 => processor1,
                                                     2 => processor2,
                                                     3 => processor3,
                                                     4 => processor4,
                                                     5 => processor5,
                                                     6 => processor6);



  function Retype_to_psor_list_rep is new Unchecked_Conversion(
                                         source => actual_psor_list_rep,
                                         target => psor_list_rep);


  procedure Initialize
  is

    -- CODE MAY NOT BE ADDED TO THIS PROCEDURE BODY!

  begin
    defined_psors := number_of_psors;
    processor_list_AD := Retype_to_psor_list_rep(processor_list);
  end Initialize;

end Processors;
```

## PROCESS DEFINITIONS PACKAGE

```
with iMAX_Definitions;
package Process_Definitions is

  -- Function:
  --   This package provides the mechanism to create and start user
  --   processes.  Instantiation of the generic package
  --   "Process_Instance" will statically bring some of the objects
  --   into existence that make up the process.  Initialization of the
  --   process is completed dynamically by calling the
  --   "Complete_process_initialization" procedure.


  subtype deadline_type is iMAX_Definitions.deadline_scheduling_value
                               range 0 .. 2**14 - 1;

  generic
      with procedure Process_Startup;
                               -- The initial procedure to be executed
                               -- by this process.
      process_name:   iMAX_Definitions.print_name := "              ";
                               -- Used mainly for debugging purposes.
      SRO_size:       ordinal := 0;
                               -- The size in bytes of this process's
                               -- stack.  If a value of 0 is specified,
                               -- a default value will be used.
      num_OT_entries: short_ordinal := 0;
                               -- The number of entries in the process's
                               -- stack object table.  If a value of 0
                               -- is specified, a default value will be
                               -- used.
      frozen:         boolean := false;
                               -- If true, create the process in frozen
                               -- memory,
                               -- If false, create the process in normal
                               -- memory.
      service_period: short_ordinal := iMAX_Definitions.
                                         default_service_period;
      deadline:       deadline_type := iMAX_Definitions.
                                         default_deadline;
      priority:       short_ordinal := iMAX_Definitions.
                                         default_priority;
```

```
 package Process_Instance is

   -- Function:
   --   This package provides the instantiator with a procedure that
   --   will complete the initialization of the process.  It also
   --   provides the process ID of the process.  This process ID
   --   is only meaningful after the process is completely
   --   initialized, i.e. after "Complete_process_initialization" is
   --   called.


   process_id:  short_ordinal;  -- The unique process ID of the
                                -- created process.

   procedure Complete_process_initialization;

     -- Function:
     --   Completes initialization of the process.

 end Process_Instance;

end Process_Definitions;
```

## USER PROCESSES PACKAGE BODY EXAMPLE #1

```
--!! Note:  This example is for the minimal configuration of iMAX.  It
--!!        is included with iMAX as the file "V1USRP.MBS".


with Process_Definitions, iMAX_Definitions;
package body User_Processes is

  -- Function:
  --   This is an example User_Processes package body which users
  --   should follow in developing their own.
  --
  --   This module instantiates the user processes and provides an
  --   Initialize procedure which completes the initialization of
  --   each user process.
  --
  --   For each user process, there is an initial procedure and an
  --   instantiation of Process_Definitions.Process_Instance which
  --   specifies the process id, the initial procedure, and the
  --   process' stack SRO and object table sizes.
  --
  --   Finally, the initialize procedure provided by this module
  --   simply calls the Complete_process_initialization procedure
  --   in each instantiation of Process_Definitions.Process_Instance.
  --
  --   This example defines two user processes which simply loop.
  --   Users may define any number of user processes to do whatever
  --   functions are needed.

  ------------------------------------
  -- DECLARATIONS FOR USER PROCESS 1 --
  ------------------------------------
  procedure UPROC1_initial_proc;  -- forward declaration

  package UPROC1 is new Process_Definitions.Process_Instance(
     Process_Startup => UPROC1_initial_proc);



  ------------------------------------
  -- DECLARATIONS FOR USER PROCESS 2 --
  ------------------------------------
  procedure UPROC2_initial_proc;  -- forward declaration

  package UPROC2 is new Process_Definitions.Process_Instance(
    Process_Startup => UPROC2_initial_proc);
  -------------------------------------------------
  -- BODY OF INITIAL PROCEDURE FOR USER PROCESS 1 --
  -------------------------------------------------
  procedure UPROC1_initial_proc
  is
    var: ordinal := 0;
  begin
    loop
      var := var + 1;
    end loop;
  end UPROC1_initial_proc;


  -------------------------------------------------
  -- BODY OF INITIAL PROCEDURE FOR USER PROCESS 2 --
  -------------------------------------------------
  procedure UPROC2_initial_proc
  is
    var: ordinal := 0;
  begin
    loop
      var := var + 1;
    end loop;
  end UPROC2_initial_proc;


  -----------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  -----------------------------------------------
  procedure Initialize
  is
  begin
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

## USER PROCESSES PACKAGE BODY EXAMPLE #2

```
--!! Note: This example is for the full configuration of iMAX.  It is
--!!       included with iMAX as the file "V2USRP.MBS".  There is only
--!!       one difference between this example and the previous one.
--!!       The example for full iMAX includes a call to the procedure
--!!       Release_Data.Output_release_data, which writes the sign-on
--!!       message "iMAX 432 V2.XX" to the system console device.

with Process_Definitions, iMAX_Definitions, Release_Data;
package body User_Processes is

  -- Function:
  --   This is an example User_Processes package body which users
  --   should follow in developing their own.
  --
  --   This module instantiates the user processes and provides an
  --   Initialize procedure which completes the initialization of
  --   each user process.
  --
  --   For each user process, there is an initial procedure and an
  --   instantiation of Process_Definitions.Process_Instance which
  --   specifies the process id, the initial procedure, and the
  --   process' stack SRO and object table sizes.
  --
  --   Finally, the initialize procedure provided by this module
  --   simply calls the Complete_process_initialization procedure
  --   in each instantiation of Process_Definitions.Process_Instance.
  --
  --   This example defines two user processes which simply loop.
  --   Users may define any number of user processes to do whatever
  --   functions are needed.

  -------------------------------------
  -- DECLARATIONS FOR USER PROCESS 1 --
  -------------------------------------
  procedure UPROC1_initial_proc;  -- forward declaration

  package UPROC1 is new Process_Definitions.Process_Instance(
     Process_Startup => UPROC1_initial_proc);


  -------------------------------------
  -- DECLARATIONS FOR USER PROCESS 2 --
  -------------------------------------
  procedure UPROC2_initial_proc;  -- forward declaration

  package UPROC2 is new Process_Definitions.Process_Instance(
    Process_Startup => UPROC2_initial_proc);
  --------------------------------------------------
  -- BODY OF INITIAL PROCEDURE FOR USER PROCESS 1 --
  --------------------------------------------------
  procedure UPROC1_initial_proc
  is
    var: ordinal := 0;
  begin
    loop
      var := var + 1;
    end loop;
  end UPROC1_initial_proc;


  --------------------------------------------------
  -- BODY OF INITIAL PROCEDURE FOR USER PROCESS 2 --
  --------------------------------------------------
  procedure UPROC2_initial_proc
  is
    var: ordinal := 0;
  begin
    loop
      var := var + 1;
    end loop;
  end UPROC2_initial_proc;


  ------------------------------------------------
  -- PROCEDURE TO INITIALIZE ALL USER PROCESSES --
  ------------------------------------------------
  procedure Initialize
  is
  begin
    Release_Data.Output_release_data;
    UPROC1.Complete_process_initialization;
    UPROC2.Complete_process_initialization;
  end Initialize;

end User_Processes;
```

# CHAPTER INI
# INITIALIZATION

This chapter describes iMAX initialization. Initialization has these parts:

- hardware initialization
- loading the central system
- software initialization
- entry to user code

This chapter does not describe initialization of Peripheral Subsystems, but presumes that software running on an Attached Processor is being used to initialize a 432 central system. Initialization of the iMAX IP Controller software is described in Chapter IOI, Input/Output Implementation.

The AP software that performs hardware initialization and loads central system memory can be either the DEBUG-432 debugger or user software. iMAX provides PL/M-86 utility routines to support users writing their own AP software to perform initialization.

Initialization using DEBUG-432 is appropriate in a development environment, for 432 software executing within the Intel 432 Cross Development System's 432/670 Execution Vehicle. Initialization via user software is required for 432 software embedded in a customer product.

## INITIALIZATION USING DEBUG-432

DEBUG-432 is the Intel 432 system debugger. DEBUG-432 executes on an Intellec Series III Microcomputer Development System as part of the Intel 432 Cross Development System (432 CDS). The Intellec unit is connected to a System 432/670 execution vehicle by an iAPX 432 Interface Processor (IP). To the 432/670 central system, the Intellec unit and its associated IP form a Peripheral Subsystem. To the user, the Intellec unit with DEBUG-432 is a means of loading, starting, observing, and controlling central system programs.

Table INI-1 lists DEBUG-432 commands used for central system initialization. These commands and other features of DEBUG-432 are described in the Intel 432 Cross Development System Workstation User's Guide. Figure INI-1 gives an example of initialization using DEBUG-432.

Table INI-1. DEBUG-432 Commands for Central System Initialization

| Command | Partial Description |
|---|---|
| INIT | places the System 432/670 in a state capable of executing programs. INIT resets the 432/670 hardware and clears the 432/670 memory. The hardware reset includes resetting all the GDPs and IPs in the System 432/670, which stops all 432 program execution. |
| INIT SYSTEM | resets the 432/670 hardware, including all the GDPs and IPs, but does not clear the 432/670 memory. Thus the memory image remain intact and ready for examination or execution. |
| DEBUG [ filename ] | enables logical addressing and optionally loads an .EOD file into the System 432/670 memory. |
| LOAD filename | loads an .EOD file into the System 432/670 memory and disables logical addressing. |
| START | starts the lowest numbered GDP in the System 432/670, which should begin executing previously loaded central system initialization code. The central system code should start all other processors in the system. The START command can be used only if logical addressing is enabled (see DEBUG). |

```
-RUN DEB432
SERIES III 432 SYSTEMS LEVEL DEBUGGER, V1.00

?include deb432.tem              -- includes Intel-supplied debugger
                                 -- "templates"

?init                            -- hardware initialization of 432/670

TOP OF MEMORY IS:  7FFFF
?debug V2test.eod                -- load the program to be debugged and
                                 -- enable logical addressing

...                              -- user can set breakpoints,
                                 -- patch memory, etc.
                                 -- before starting program

?start                           -- starts central system processing
iMAX 432 V2.XX                   -- initial message for full iMAX (where
                                 -- Xs are decimal digits giving
                                 -- additional version information)
```

Figure INI-1. Initialization Using DEBUG-432

**NOTE**

Users of DEBUG-432 who need to know the object coordinates of user processes at initialization should set breakpoints at the start of the initial procedure for each user process. DEBUG-432 will give the object coordinates of the processes in the breakpoint notification messages.

## INITIALIZATION INTERFACE FOR USER AP SOFTWARE

This section describes a PL/M-86 interface available to users writing their own Attached Processor software to initialize a 432 central system. The interface is defined by the PL/M-86 files LDUTIL.EXT and LDUSER.EXT. LDUTIL contains subroutines to initialize the 432 hardware, load a system image into 432 memory, and start a GDP. The user of LDUTIL must supply certain public variables and routines that are referenced by the load utility. These variables and routines are specified by the file LDUSER. These load utilities use the iMAX IP controller software described in Chapter IOI, Input/Output Implementation. The load utilities require the IP used to be in physical mode.

## HARDWARE INITIALIZATION

The routine Initialize_system resets the system bus, initializes the IP windows, and zeroes central system memory. The routine uses the top_of_memory variable provided by the user.

After hardware initialization, the control window (IP window 4) of the IP used for initialization is mapped to the top 256 bytes of physical memory in the central system (as determined by the top_of_memory variable), and the IP is left in physical mode. If the user wants this window mapped to some other part of the central system's memory, an ALTER MAP operation can be done using the iMAX IP Controller software. The iMAX IP Controller software is described in Chapter IOI, Input/Output Implementation.

The hardware initialization sequence resets all 432 processors in the system.

If more than one AP executes the hardware initialization sequence, then the last AP to perform the sequence will force all IPs in the system into their initial state, in physical mode with the default window settings.

## LOADING

Two routines are provided for loading a system image. Load LIF supports the load image file (.LIF) format which is created using the DEBUG-432 SAVE command. Load block copies a block of AP memory to a specified location in 432 memory, supporting user-defined formats. Both system loading routines use IP window 0 in buffered mode.

### Load LIF

The DEBUG-432 SAVE command produces an ISIS file in LIF format, which can be loaded by calling Load_LIF. Load_LIF makes requests to the user-provided routine Read_LIF_data for buffers containing portions of the LIF. Load_LIF has no parameters.

### Load block

Support for user-defined load formats is provided by the routine Load_block. Load_block can also be used in combination with Load_LIF. Load_block copies the specified buffer of data to central system memory at the specified address. Load_block has four parameters, the 432 memory address in two parts (high_432_addr, low_432_addr), the length of the buffer (length, actually length in bytes minus one), and a pointer to the buffer in AP memory (buffer_ptr). The maximum buffer length is 2**16 (65,536) bytes.

## STARTING THE SYSTEM

After the system image has been loaded, the Start_GDP routine can be called to start the system. iMAX initialization assumes that the AP will start only one GDP. Any other GDPs in the system will be started by iMAX code running in the central system.

**WARNING**

If more than one GDP is started by AP software, the effects are unpredictable and certainly undesirable.

Start_GDP has one parameter, the processor ID of the GDP to be started (processor_ID). The routine sends a Start IPC to the specified GDP and returns after the IPC operation completes. Start_GDP does not wait for acknowledgement from the 432.

**WARNING**

If there is a processor object in central system memory, but no physical processor with the specified processor ID, then the IPC will complete successfully, but the system will not start.

## USER-PROVIDED INFORMATION

The iMAX utility routines described require an environment containing a public variable and one or more routines provided by the user.

The user-defined record top_of_memory must contain the address of the last byte of central system memory. This value is used for zeroing system memory and for locating the IP control window.

The user-defined procedure User_error is called when an error occurs during initialization. User_error has one parameter (error_msg_ptr), a pointer to a null-terminated ASCII string containing an error message. Table INI-2 lists the possible error messages.

If Load_LIF is used to load the system, then the user must supply the procedure Read_LIF_data to read the system image one buffer at a time. Read_LIF_data has two parameters: a pointer to a buffer where the LIF data should be placed (buffer_ptr), and a requested data length (length, which is actual length, not length minus one).

Table INI-2. AP Initialization Utilities Error Messages

| Message | Comments |
|---|---|
| IP Context fault | The IP has faulted. |
| IP Process fault | " " " |
| IP Processor fault | " " " |
| IP Fails to perform requested function (timed out) | The IP fails to respond. |
| Failed to initialize 432 system | The hardware INIT signal was not acknowledged. |
| Checksum discrepancy on LIF record | There is an error in the LIF file to be loaded. |
| Bad LIF record type detected | " " " |
| Bad header/LLA records on LIF | " " " |
| Object Descriptor Map Failure | Start_GDP failed due to an error in the program image loaded into the central system. |
| Bad Object Descriptor | " " " |
| Bad Object Descriptor System Type | " " " |
| Bad Object Descriptor Processor Type | " " " |
| Access Descriptor Map Failure | " " " |
| AD not valid | " " " |
| Work window invalidation failure | " " " |
| Send IPC failure | The start IPC failed because the target PCO was locked. |

## CENTRAL SYSTEM PROGRAM INITIALIZATION

At this point in the initialization sequence, hardware initialization is complete, a user program containing iMAX has been loaded, and one GDP has been started. This GDP now executes iMAX initialization code. This code is different for the two iMAX configurations -- full iMAX and minimal iMAX. The initialization code creates additional system objects needed by iMAX, starts support processes internal to iMAX (such as the parallel garbage collection process), and starts any other GDPs and IPs in the system.

## INITIAL SYSTEM MESSAGES

The minimal configuration of iMAX writes the messages listed in Appendix MIN, Minimal Configuration. The full configuration of iMAX writes the message: "iMAX 432 V2.XX", where the Xs are decimal digits giving additional version information.

## ENTRY TO USER CODE

The first user-supplied modules to execute are the procedures IP_Processes.Initialize (first) and User_Processes.Initialize (second), with specification provided by iMAX and body modified or supplied by the user.

The body of IP_Processes.Initialize creates connections, creates IP processes, and starts IPs. A default body is supplied by iMAX to initialize iMAX I/O. Users modify this body if they change the I/O configuration by adding or removing devices or Peripheral Subsystems. The IP_Processes package is described in Chapter IOI, Input/Output Implementation.

The body of User_Processes.Initialize should call Complete_process_initialization for any static processes defined by the user. The user-supplied body can also use iMAX Basic_Process_Management to create and start dynamic processes. The User_Processes package and static processes are described in Chapter CON, Configuration.

## LDUTIL.EXT LISTING

```
/**********

        Title:  LDUTIL - System loading utilities.

        Function:  The routines declared externally here provide the
                   user with the ability to initialize, load, and start
                   a 432 system.  Note that to use them, a user must
                   first provide definitions for the externals listed in
                   LDUSER.

                   These routines make use of the IP jumpered to
                   Multibus addresses 80000H - 8FFFFH.  Unless otherwise
                   stated these routines assume the IP is in physical
                   mode and leave it in physical mode.

**********/

/**********

     INITIALIZE SYSTEM:  This procedure initializes the 432 system
         and zeroes all memory up to top_of_memory. The 432 system can be
         in any state when this is called (as long as it is powered up),
         and the IP used to perform initialization will be left in
         physical mode with its control window open on the upper 256
         bytes of memory.

**********/

Initialize_system:
  PROCEDURE EXTERNAL;
  END
Initialize_system;


/**********

     LOAD LIF:  This procedure loads an LIF (load image file) formatted
         byte stream into the 432 system. It gets its data via the user-
         provided procedure Read_LIF_data.

**********/

Load_LIF:
  PROCEDURE EXTERNAL;
  END
Load_LIF;
```

```
/**********

    LOAD BLOCK:  This procedure loads the specified block of data into
        the 432's memory. A block cannot be longer than 64K bytes.

        The length parameter is one less than the actual number of
        bytes.

**********/

Load_block:
  PROCEDURE(high_432_addr, low_432_addr, length, buffer_ptr) EXTERNAL;
    DECLARE
      high_432_addr   BYTE,         /* upper 8 bits of 432 address */
      low_432_addr    WORD,         /* lower 16 bits of 432 address */
      length          WORD,         /* one less than actual byte count */
      buffer_ptr      POINTER;      /* starting address of source */
  END
Load_block;


/**********

    START GDP:  This procedure starts the specified GDP by sending
        it a "start" IPC. It assumes that an image has been already
        loaded and that the specified GDP exists. If the GDP processor
        object exists, but no corresponding physical GDP is plugged in,
        the IPC will be successfully sent (though never received), and
        no user error message will be generated.

**********/

Start_GDP:
  PROCEDURE(processor_id) EXTERNAL;
    DECLARE
      processor_id    BYTE;     /* ID of GDP processor to start */
  END
Start_GDP;
```

## LDUSER.EXT LISTING

```
/**********

    Title:  LDUSER - user defined load externals.

    Function:  Users of the system load utilities must provide
               definitions for these externals.

**********/


 /**********
     The top 432 memory address must be known in order to initialize
     memory. The user must declare top_of_memory as a PUBLIC variable.
 **********/

DECLARE top_of_memory
  STRUCTURE(low    WORD, /* lower 16 bits of 432 address */
            high   WORD) /* upper 8 bits of 432 address (left padded) */
    EXTERNAL;


/**********

    READ LIF DATA:  This procedure is called by the Load_LIF utility.
        It is expected to get the next specified number of bytes
        from an LIF (load image file) formatted byte stream and put
        them into the specified buffer.

**********/

Read_LIF_data:
  PROCEDURE(buffer_ptr, length) EXTERNAL;
    DECLARE
      buffer_ptr     POINTER,  /* starting address of buffer    */
      length         WORD;     /* byte count to place in buffer */
  END
Read_LIF_data;
```

```
/**********

    USER ERROR:  This procedure is called whenever a user error is
        detected in the load utility routines (e.g. if a bad LIF record
        is encountered). The error_msg_ptr points to an error message
        consisting of ASCII bytes terminated by a zero byte (ASCII
        null).

        In general, user errors are fatal and this routine should not
        return control.

**********/

User_error:
  PROCEDURE(error_msg_ptr) EXTERNAL;
    DECLARE
      error_msg_ptr  POINTER;  /* points to error message string */
  END
User_error;
```

This appendix lists the software components that make up iMAX 432. These components are divided as follows:

1. VAX host system files (VMS or UNIX operating systems), supplied on magnetic tapes (600', 1600 bpi):

    a. Ada package specifications and environments for user-visible packages.
    b. Ada package bodies modifiable by users (and corresponding environments).
    c. already linked .EOD (external object description) files that contain iMAX configurations for the user to link to.

2. Intellec Series III Microcomputer Development System files (ISIS-II operating system) supplied on floppy diskette (either single or double-density):

    a. PL/M-86 specifications for user-visible interfaces to iMAX IP controller software (.EXT files).
    b. PL/M-86 modules modifiable by users, and corresponding object files (.PLM, .OBJ files).
    c. already linked .LNK files that contain iMAX IP controller or terminal support software
    d. ICU88 configuration files to support users configuring AP software.

For each software component listed below, the file name and extensions are given, along with the name of any Ada packages defined by the file, and a reference to the chapter (if any) where the listing can be found.

The Ada package TEXTIO is supplied with the iMAX host files. However, TEXTIO is not part of iMAX and the files associated with TEXTIO are not listed here. TEXTIO is described in the Ada Support Packages User's Guide.

Users of the UNIX operating system on the VAX host must specify iMAX file names in upper case (UNIX is case-sensitive). User's of the VMS operating system can use upper case, lower case, or a mixture of cases in file names (VMS is not case-sensitive).

These extensions are used for the VAX host files:

| | |
|---|---|
| .MLS | listing of the specification for an iMAX module |
| .MBS | source file for the body of an iMAX module (the body can be modified by the user) |
| .MLE | environment file for an iMAX module |
| .MLI | integrated environment file for one or more iMAX modules, produced by the COMBINE utility |
| .EOD | linked "external object description" for one or more modules, or for an entire iMAX configuration |

The VAX host file extensions follow conventions described in the Intel 432 Cross Development System VAX Host User's Guide.

VAX Host Files

| Name | Extensions | Description | Chapter |
|---|---|---|---|
| DESDEF | .MLS, .MLE | Descriptor_Definitions | DEF |
| MAXDEF | .MLS, .MLE | iMAX Definitions | DEF |
| SRO | .MLS, .MLE | SRO_Manager | STO |
| MEMCTL | .MLS, .MLE | Memory_Controller | STO |
| BPM | .MLS, .MLE | Basic_Process_Management | BPM |
| PORTS | .MLS, .MLE | Untyped_Ports | COM |
| TYPORT | .MLS, .MLE | Typed_Ports | COM |
| CXDEFS | .MLS, .MLE | Context_Definitions | PEN |
| PGLOB | .MLS, .MLE | Process_Global_Definitions | PEN |
| EXTMGT | .MLS, .MLE | Extended_Type_Manager | EXT |
| IODEFS | .MLS, .MLE | IO_Definitions | IO |
| IOINTF | .MLS, .MLE | Synchronous_IO_Interfaces | IO |
| TERMIN | .MLS, .MLE | Terminal_Interfaces | IO |
| DBISRC | .MLS, .MLE | Debug_Source | IO |
| DBISNK | .MLS, .MLE | Debug_Sink | IO |
| IODEV | .MLS, .MLE | IO_Devices | IO |
| DEVINT | .MLS, .MLE | Asynchronous_IO_Interface | IO |
| IPPRCS | .MBS, .MLI | IP_Processes body | IOI |
| TSRC | .MLS, .MLE | Actual_Terminal_Sources | IOI |
| TSINK | .MLS, .MLE | Actual_Terminal_Sinks | IOI |
| IPMAN | .MLS, .MLE | IP_Management | IOI |
| PSORS | .MBS, .MLI | Processors body | CON |
| PROC | .MLS, .MLE | Process_Definitions | CON |
| V1USRP | .MBS, .MLI | User_Processes body (minimal iMAX example) | CON |
| V2USRP | .MBS, .MLI | User_Processes body (full iMAX example) | CON |
| MINMAX | .EOD | linked minimal iMAX configuration | --- |
| IMAX | .EOD | linked full iMAX configuration | --- |

These extensions are used for the ISIS-II files:

| | |
|---|---|
| .INC | files to be included in PL/M-86 modules. Each "include file" may declare a set of related literals and external procedures. |
| .EXT | external declarations of iMAX-supplied or user-supplied interfaces (equivalent to an Ada package specification) |
| .PLM | code and data for a PL/M-86 module (equivalent to an Ada package body) |
| .A86 | code and data for an ASM-86 assembly language module |
| .OBJ | object code and data corresponding to a PL/M-86 or ASM-86 module |
| .LNK | linked file corresponding to one or more AP program modules |
| .CON | ICU88 configuration file |
| .CSD | ISIS-II command file |
| [null] | used in this appendix to mean no extension |

PL/M-86 files, configuration files, and linked files all have two versions in the list below, for both the non-megabyte and megabyte versions of iRMX 88 (except for LDUTIL.LNK, which is the same for both versions). The first files in the list are for the non-megabyte version. The following files (with "M" as the first letter of their file names) are for the megabyte version.

ISIS-II Files

| Name | Extensions | Description | Chapter |
|---|---|---|---|
| APEXEC | .INC | AP_Executive_Calls | IOI |
| BASDEF | .INC | IP_Basic_Definitions | IOI |
| FUNCTN | .INC | IP_Function_Interface | IOI |
| XFER | .INC | IP_Transfer_Driver | IOI |
| FRMGR | .INC | IP_Function_Manager | IOI |
| WINDOW | .INC | IP_Window_Manager | IOI |
| APINI0 | .EXT | AP_Initialization utilities -- no terminal | IOI |
| APINI1 | .EXT | AP initialization utilities -- one terminal | IOI |
| APINI5 | .EXT | AP initialization utilities -- five terminals | IOI |

ISIS-II Files

| Name | Extensions | Description | Chapter |
|---|---|---|---|
| INITL0 | .PLM, .OBJ | AP initialization task -- one terminals | IOI |
| INITL1 | .PLM, .OBJ | AP initialization task -- one terminal | IOI |
| INITL5 | .PLM, .OBJ | AP initialization task -- five terminals | IOI |
| TERM0 | .CON, .CSD, .A86, [null] | ICU88 files -- no terminals | --- |
| TERM1 | .CON, .CSD, .A86, [null] | ICU88 files -- one terminal | --- |
| TERM5 | .CON, .CSD, .A86, [null] | ICU88 files -- five terminals | --- |
| LDUTIL | .EXT, .LNK | AP load utilities | INI |
| LDUSER | .EXT | user-supplied information for AP load utilities | INI |
| IPCTRL | .LNK | linked IP controller (and iRMX 88) | --- |
| SUP1TH | .LNK | linked support for one terminal | --- |
| SUP5TH | .LNK | linked support for five terminals | --- |

/* Remaining files are equivalent to ones above, but use the <u>megabyte</u> version of iRMX 88.*/

| Name | Extensions | Description | Chapter |
|---|---|---|---|
| MINIT0 | .PLM, .OBJ | AP initialization task -- no terminal | IOI |
| MINIT1 | .PLM, .OBJ | AP initialization task -- no terminal | IOI |
| MINIT5 | .PLM, .OBJ | AP initialization task -- five terminals | IOI |
| MTERM0 | .CON, .CSD, .A86, [null] | ICU88 files -- no terminal | --- |
| MTERM1 | .CON, .CSD, .A86, [null] | ICU88 files -- one terminal | --- |
| MTERM5 | .CON. ,CSD, .A86, [null] | ICU88 files -- five terminals | --- |
| MIPCTL | .LNK | linked IP controller (and iRMX 88) | --- |
| MSUP1T | .LNK | linked support for one terminal | --- |
| MSUP5T | .LNK | linked support for five terminals | --- |

# APPENDIX MIN
# MINIMAL CONFIGURATION

This appendix describes minimal iMAX, a subset of full iMAX that provides a simple environment for executing multiple processes on multiple 432 processors.

Minimal iMAX does not provide storage reclamation or compaction, run-time process management, extended types, or fault handling. Minimal iMAX does not support 432 Interface Processors and only provides I/O through the DEBUG-432 debugger.

The user-visible packages in minimal iMAX are listed below. With the exceptions noted, these modules are the same as the corresponding modules in full iMAX, and are described in the main part of this manual.

Table MIN-1. Minimal iMAX Packages

```
Descriptor_Definitions
iMAX_Definitions

Untyped_Ports
Typed_Ports

Context_Definitions
Process_Globals_Definitions
  Note:  In minimal iMAX process global access segments, these
         fields are null: context_fault_area,
         extended_type_manager, simulator, and process_manager.
         The Create_local_heap function referenced by the
         sro_manager field always returns an access for the single
         global heap SRO in a minimal configuration. The
         default_global_heap_sro field also references the single
         global heap SRO. The standard_input, standard_output,
         and standard_error I/O interfaces are all assigned to the
         DEBUG-432 console.

IO_Definitions
Synchronous_Interfaces
Debug_Source   -- Get_asynchronous_interface returns null.
Debug_Sink     -- Get_asynchronous_interface returns null.

Process_Definitions
User_Processes
Processors
  Note:  The processor list may contain only GDP objects (no IP
         objects) in minimal iMAX.
```

The only processes active in a minimal iMAX system are those created at initialization as user static processes (as described in Chapter CON, Configuration).

A minimal iMAX system contains a single global heap SRO. Storage allocated from this SRO is never reclaimed.

## ADA PROGRAMMING RESTRICTIONS

This section lists features of the Ada programming language that minimal iMAX cannot support and suggest ways that users can "program around" these missing features. In general, users needing capabilities not supported by the minimal configuration should use the full configuration of iMAX. Some of the features listed that minimal iMAX cannot support are not yet implemented by the 432 Ada compiler in any case. A list of Ada features not yet implemented by the 432 Ada compiler is contained in the _Intel 432 Cross Development System VAX Host User's Guide_.

| Restriction and Suggested Action | Explanation |
|---|---|
| Packages and access types cannot be declared within a subprogram.<br><br>Declare all packages and access types outside of subprograms. | Packages or access types local to a subprogram require creation of local heap SROs, which are not supported by minimal iMAX. |
| Ada tasks cannot be supported (and neither can iMAX dynamic processes).<br><br>Use iMAX static processes instead of Ada tasks or iMAX dynamic processes. Operations available on static processes include _Idle_ and the interprocess communication operators. | Minimal iMAX does not include the process management and storage management features needed to support Ada tasks (or iMAX dynamic processes). |
| A subprogram call cannot create a context access segment or a context data segment larger than 2,048 bytes (512 access descriptors). Variables declared local to a subprogram cannot contain any segment larger than 2,048 bytes. | The 432 GDP will zero a newly created segment only if it is not larger than 2,048 bytes. If a larger segment is created from "dirty" (non-zero) memory, the GDP faults, and software must zero the segment. Minimal iMAX does not provide the fault handler to do this zeroing. |

| Restriction and Suggested Action | Explanation |
|---|---|
| Either:<br>1) Break up the subprogram into smaller subprograms, or<br>2) Replace large local variables with access variables referencing segments allocated with the Ada new operator from the global heap SRO. | Because minimal iMAX initialization zeroes all free memory, and because segments created from the single global heap SRO are never reclaimed, any segment created from the global heap in minimal iMAX is created from "clean" (all zeroes) memory and can be larger than 2,048 bytes without causing a fault. However, because stack memory is constantly reused, it will almost certainly be "dirty" and any attempt to create a segment larger than 2,048 bytes from the stack will almost certainly fault. |

## CONSOLE MESSAGES

This section lists messages that appear or may appear on the DEBUG-432 terminal when a minimal iMAX system initializes. Note that minimal iMAX is not a proper subset of full iMAX in this regard; only the first message appears for full iMAX initialization.

All numbers given as a string of "d"s (e.g., "ddd") are displayed in variable-length decimal fields.

| Message | Description |
|---|---|
| iMAX432 V2.XX | -- Initial message (where Xs are decimal digits giving additional version information) |
| Error: MEMTOP = dddddd bytes, but dddddd bytes are required. | -- Written only if the MEMORYTOP directive to LINK-432 did not specify enough memory to hold the load image plus objects created at initialization. |
| MEMTOP rounded up to dddddd bytes. | -- Written only if the MEMORYTOP value specified to LINK-432 was not a multiple of 8, in which case MEMORYTOP is rounded up to the next multiple of 8 and the new value is printed. |

| Message | Description |
|---|---|
| Error: Not enough free descriptors in object table directory. | -- Written only if directives to LINK-432 did not specify enough OTD descriptors for the number of static processes in the system. |
| Global heap SRO size = dddddd bytes | -- Number of free bytes available in the system's single global heap SRO. Note that this message is written after the allocation of IP processes and user static processes from the global heap. |
| Processor ddd dispatching | -- Written for each GDP that is started. The processor ID is printed. |
| Process ddd started, coordinates: hh^hh SRO size = ddd bytes, OT size = ddd descriptors | -- Written for each process that is started. The numeric process ID is printed. Process object coordinates are in hexadecimal. SRO size is the total size of the process stack SRO's allocation block. OT size is the total number of entries in the process object table (including the header entry). |
| Error: Process ddd terminated. | -- Written when a process is terminated because it returns from its initial context. |

# APPENDIX RGT
# USER-VISIBLE TYPE RIGHTS

Type rights are access rights defined for a particular type of 432 object. Type rights are used to restrict the operations possible using a particular access. For example, an access for a 432 object of type port might have send rights but not receive rights. A context with such an access can only send a message to the port using the access, but cannot receive messages from the port using it.

There are three type rights bits on any access descriptor, named type right 1, type right 2, and type right 3. When a type right is interpreted for a particular object type, it is renamed. For example, send rights renames type right 1 for port accesses.

This appendix lists all the 432 object types that users can access using iMAX, and describes the type rights defined for each type. Object types internal to iMAX and not accessible to users (e.g., object tables, processor objects) are not listed. The first type right listed for an object type renames type right 1, the next renames type right 2, etc. Any remaining type rights bits are not interpreted by iMAX.

| System Object Type | Type Rights | Required To: |
|---|---|---|
| storage_resource | create_rights | create an object or refinement using a storage resource |
| port | send_rights | send a message or forward a carrier to a port |
| | receive_rights | receive a message from a port |
| carrier | use_rights | use a carrier in a surrogate operation |
| domain | ----- | ----- |

| System Object Type | Type Rights | Required To: |
|---|---|---|
| type_definition | create_rights | create an instance of an extended type |
| | retrieve_rights | retrieve the representation of an extended_type object |
| process | control_rights | start, stop, or destroy a process and/or its descendants |

This appendix describes how iMAX handles (or doesn't handle) faults, Ada exceptions, system errors, and trace events; and how iMAX reports these conditions to the user or to user software. iMAX V2 provides different fault-handling for the minimal and full configurations.

This appendix does not list detailed fault information. Detailed information on 432 faults is found in the iAPX 432 General Data Processor Architecture Reference Manual.

## FAULT LEVELS

The 432 architecture defines three severity levels for faults:

- context faults are the least severe and cause a transfer of control to a context fault handler instruction object. The context fault handler is not part of iMAX and is supplied by a programming language run-time environment, or (less commonly) by a user.

- process faults are of greater severity; a process fault stops the running process and sends it as a message to a port. An iMAX fault handling process receives the faulted process. iMAX may restart the process transparent to the user; may vector the fault downward to the faulting context's context-level fault handler; may send the faulting process to its guardian port; or may report the error via the DEBUG-432 debugger or via an output device.

- processor faults are the most severe; a processor fault causes the faulting processor to suspend execution of its current process (if any) and enter a special diagnostic dispatching mode in which it dispatches a process found at a diagnostic dispatching port. iMAX handles all processor-level faults. iMAX may vector the fault downward to process-level, or may report the error via the DEBUG-432 debugger or via an output device.

If fault-handling fails at a particular level, the fault is propagated to the next higher level. For example, a context fault within a context fault handler causes a process fault. If a processor fault cannot be handled by the hardware (e.g., the diagnostic port descriptor is invalid), then the processor halts and asserts an external hardware error signal.

Context-level faults reflect an error in a user program, e.g., arithmetic overflow faults. Some faults handled at the context-level are reported by the architecture as process-level faults that iMAX converts to context-level. For example, addressing past the bounds of a segment, type errors, and rights errors are all kinds of faults that are converted to context-level by iMAX.

## MINIMAL iMAX FAULT-HANDLING

Minimal iMAX does not provide any fault-handling capabilities. Faults normally cause the DEBUG-432 debugger to display a fault message at the debug console. The only exceptions are certain "unannounced faults" that DEBUG-432 is unable to respond to. How DEBUG-432 handles faults, and how users can detect unannounced faults, are described in the Intel 432 Cross Development System Workstation User's Guide.

## FULL iMAX FAULT-HANDLING

Full iMAX provides extensive fault-handling, transparent to users. Faults handled transparently by full iMAX include:

- A newly allocated segment larger than 2,048 bytes must be zeroed.
- A segment is temporarily not addressable due to memory management operations.
- Additional object table space or free physical memory is required to satisfy a request to create an object.
- A Return instruction causes a fault so that a local heap SRO and objects allocated from it can be reclaimed.

A context fault or process fault not handled by iMAX has the following result:

- Dynamic processes (defined using iMAX Basic_Process_Management) are sent to their guardian port in condition error, system_error, or fatal_error.

- Static processes are sent to the DEBUG-432 debugger and cause a fault message to be displayed at the debugger console.

A processor fault not otherwise handled by iMAX triggers a system error (as described below in the section System Errors).

## ADA EXCEPTIONS

The 432 Ada Compiler System has not yet implemented the Ada exception-handling facility. Wherever this manual describes iMAX raising an exception, iMAX actually calls an internal iMAX procedure that causes the process in which the exception occurs to fault. The resulting fault is handled as follows:

- Dynamic processes (defined using iMAX Basic_Process_Management) are sent to their guardian port in state (error, *, true, no_exception_handler, no_service_needed, false).

- Static processes are sent to the DEBUG-432 debugger and cause a fault message to be displayed at the debugger console.

## SYSTEM ERRORS

iMAX defines a class of errors called system errors which should not occur. A system error usually indicates an error in iMAX. If a system error occurs, iMAX attempts to write error information to the current iMAX console device with this prefix:

```
**** SYS ERROR ****
****   CRASH   ****
```

iMAX then halts the system by stopping all processors (both GDPs and IPs). If a system error occurs, users should contact Intel, and provide the information displayed by the system error message as well as any other useful information.

## TRACE EVENTS

The DEBUG-432 debugger provides the handler for trace events used by the 432 Ada compiler (and thus used in all iMAX systems). DEBUG-432 supports user-defined breakpoints as well as the trace modes defined by the 432 architecture. DEBUG-432 is described in the Intel 432 Cross Development System Workstation User's Guide.

This appendix describes the hardware configuration requirements for iMAX. Additional Intel 432 hardware information is contained in the System 432/600 System Reference Manual.

## COMPONENTS

The central system portion of iMAX requires Release 2.1 General Data Processors.

The iMAX AP software requires Release 2.1 Interface Processors.

## PROCESSOR CONFIGURATION

The user must specify to iMAX the number of GDPs and IPs that iMAX must support, and also specify each processor's ID. How to do this is described in Chapter CON, Configuration.

The physical processor configuration must match the configuration specified to iMAX, except that fewer physical GDPs may be present than are specified to iMAX.

For 432/600 systems there is a correspondence between processor ID and the bus slot in which the processor card is installed (see Table HDW-1).

Table HDW-1. Processor ID Number to Bus Slot Mapping for 432/600 Systems

| 6-Slot System Bus Backplane | | 12-Slot System Bus Backplane | | 18-Slot System Bus Backplane | |
|---|---|---|---|---|---|
| SLOT | ID | SLOT | ID | SLOT | ID |
| 4 | 1 | 8 | 1 | 12 | 1 |
| 5 | 2 | 9 | 2 | 13 | 2 |
| 6 | 3 | 10 | 3 | 14 | 3 |
| | | 11 | 4 | 15 | 4 |
| | | 12 | 5 | 16 | 5 |
| | | | | 17 | 6 |

The default processor configuration in the System 432/670 Execution Vehicle (which has a 12-slot system bus backplane) is:

| Slot | Board | Processor ID |
|---|---|---|
| 8 | GDP | 1 |
| 9 | GDP | 2 |
| 10 | --- | 3 |
| 11 | IPL (for debugger) | 4 |
| 12 | IPL (for AP system) | 5 |

Note: IPL = Interface Processor Link board

The iMAX-supplied body of the Processors package is configured for this default configuration.

## INTERCONNECT ADDRESS SPACE

iMAX makes only minimal assumptions about the contents of the interconnect address space. The iAPX 432 architecture and iMAX have these requirements:

Interconnect physical address 0 must return a double-byte with the low byte containing the processor ID of the requesting processor. This double-byte can be Read-only.

Interconnect physical address 2 must be a Read/Write double-byte local to each processor supporting IPCs (Inter-Processor Communications) as described in the System 432/600 System Reference Manual.

Intel's 432/600 products meet both of these requirements and also supply additional interconnect registers. IMAX does not use any of the additional registers. Note that the processor ID and IPC registers are required for both GDPs and IPs.

## JUMPERS

This section describes required jumper settings for the proper operation of iMAX V2 AP software. This information does not apply to the minimal iMAX configuration (because there is no AP software associated with minimal iMAX).

The following hardware jumpers on the iSBC 86/12A and the iSBC 534 boards are assumed by the AP software. Unspecified jumpers should be left at their factory settings.

1. SINGLE-TERMINAL VERSION

   A. On the 86/12A board:
      1. jumper E69 - E77 -- interrupt 4 from the MULTIBUS
      2. jumper E75 - E82 -- interrupt 6 from 86/12 serial port receive-ready
      3. jumper E74 - E90 -- interrupt 7 from 86/12 serial port transmit-ready

2. MULTIPLE-TERMINAL VERSION

   A. On the 86/12A board:
      1. jumper E69 - E77 -- interrupt 4 from the MULTIBUS
      2. jumper E68 - E76 -- interrupt 5 from the MULTIBUS
      3. jumper E75 - E82 -- interrupt 6 from 86/12 serial port receive-ready
      4. jumper E74 - E90 -- interrupt 7 from 86/12 serial port transmit-ready

   B. On the 534 board:
      1. For the wire net of each serial I/O port, jumper: 6 to 7, 8 to 10, and 9 to 11.
      2. Remove jumper 118 - 125; jumper 119 - 125. -- changes the base address to 0A0H ($160_{10}$).
      3. Remove jumper 132 - 140. -- disconnect PIC 1.
      4. Remove jumper 131 - 140; jumper 131 - 136. -- connect PIC 0 to MULTIBUS interrupt 5.

# APPENDIX PRS
# PROCESS SCHEDULING INFORMATION

This appendix guides users in choosing process scheduling parameters. The process scheduling parameters referred to in this appendix are described in Chapter BPM, Basic Process Management.

## SYSTEM PROCESS SCHEDULING INFORMATION

This section provides information about the scheduling parameters of system processes internal to iMAX. These system processes compete with user processes for scheduling. User processes with too high a priority can effectively lock out iMAX system processes. Users should follow the recommendations in this section when assigning their process scheduling parameters, or risk an interruption of iMAX services in their system.

The restrictions in this section do not apply to minimal iMAX systems, because there are no system processes after initialization in minimal iMAX systems.

System processes use priorities 10 and above. User processes should normally use priorities in the range 0..10. User processes with priority 10 compete on an equal priority with several system processes for dispatching. User processes with priority 10 should have reasonable service_period and deadline parameter values to allow system processes to run. The default process tuning values (described in Chapter DEF, Basic Definitions) are safe to use.

User processes with priorities higher than 10 should only be used for low-level time-critical processing. They should be self-limiting (i.e., normally wait at a port for work).

User processes should never use priority 65535 (the highest possible priority), which is reserved for critical system processes.

## CHOOSING SCHEDULING PARAMETERS

This section provides guidance in selecting process scheduling parameters. The service and deadline parameters are specified in terms of system time units. For 432/600 systems, a system time unit is 256 microseconds.

The service parameter should be set high enough to reduce the overhead of process redispatching, and often high enough to allow the process to complete one or more transactions (or whatever its particular unit of work is) in a single service period. The service parameter should be set low enough to avoid impairing system response by blocking higher-priority processes for an undue time. In deciding how to set the service parameter to attain a particular level of responsiveness in a system, the user must also consider the number of processors in the system. For example, with 4 GDPs and a large number of compute-bound processes, the service parameter could be 40 milliseconds for all the processes and some GDP would still arrive at the dispatching port looking for work every 10 milliseconds on the average. Typical values of service are in the range .01 to .1 seconds (40 to 400 system time units for 432/600 systems).

In many applications, the periods parameter can be set to $2^{16} -1$ (65,535) and process scheduling parameters can be fixed for the life of a process. For applications that dynamically tune scheduling parameters, typical values of periods are in the range 100 to 1000. One use of periods is to periodically return processes to their owners to enforce time limits on the execution of any given process, as is done in many multi-user systems. The maximum number of system time units that a process will execute before being sent to its guardian (if periods is not equal to $2^{16} -1$) is: periods * service.

The deadline parameter can be used to reduce the time that a process spends waiting to be dispatched compared to other processes of the same priority. Consider, for example, two processes with the same priority and service period, A and B. A is compute-bound and always runs for its full service period. B normally blocks after running a short time and must be redispatched. If B is given a deadline of 0 and A a deadline of 1000, then B will normally run ahead of A when both are waiting to be dispatched, and B will more likely get a fair share of processor resources.

The priority parameter is most important of all and should be used cautiously, because higher priority processes that do not block can "starve" other processes in the system. It is even possible to starve the support processes in iMAX, and user processes should normally have the same or lower priority than these support processes, as suggested in the previous section. In contrast, the iMAX support processes will never starve user processes, because they are designed to limit their own consumption of processor time by either idling or blocking at communication ports for work.

# APPENDIX SIZ
# SIZE INFORMATION

This appendix provides information about iMAX memory requirements. This information is presented for the use of system designers who need to estimate memory requirements for systems using iMAX.

Table SIZ-1 gives the basic 432 memory requirement for the minimal and full configurations of iMAX. The 432 memory requirement for each configuration includes a static requirement and a dynamic requirement. The static requirement is the size of the load image that is loaded into 432 memory by the initializing Peripheral Subsystem. The dynamic requirement consists of objects needed by iMAX that are dynamically allocated at system initialization.

Table SIZ-1. Basic iMAX Memory Requirements

| | STATIC | DYNAMIC | TOTAL |
|---|---|---|---|
| Minimal iMAX | 47K | 0K | 47K |
| Full iMAX | 233K | 47K | 280K |

Notes:
1) Figures are approximate
2) K = 1,024 bytes
3) Memory requirements for iMAX V2 functionality will be less in future iMAX releases.
4) The static memory requirement for both configurations includes about 8K bytes needed to support the DEBUG-432 debugger.

In the full configuration of iMAX, some of the iMAX code and data structures that is used only for system initialization are reclaimed by iMAX garbage collection. This reclamation makes some of the total memory required by full iMAX available to the user after initialization. The reclamation of initialization structures also means that an iMAX system cannot be reinitialized without reloading the central system.

Table SIZ-2 gives the 432 storage requirement for each instance of particular iMAX objects that are created. iMAX objects can require more memory than the corresponding architecture-defined system objects because iMAX can extend system objects to include additional information. One such object may actually be a cluster of several segments of varying types. For example, an iMAX process includes a process carrier, a process globals access segment, and a process object table in addition to the process access segment and process data segment. Both the number of bytes required for any segments and the number of object descriptors required are tabulated, with a total of the bytes used by all segments and object descriptors. The "Bytes in Segments" figure includes the eight-byte segment header and any pad bytes required at the end of the segment, but does not include the 16 bytes required for the object descriptor for the segment.

Table SIZ-2. Sizes of Selected iMAX System Objects (bytes)

| Object Type | Bytes in Segments | Object Descriptors | Total Bytes |
|---|---|---|---|
| GDP Processor Object | 496 | 11 | 672 |
| GDP Process Object<br>-- does not include:<br>-- initial context,<br>-- process stack,<br>-- free ODs in process<br>-- object table | 392 | 7 | 504 |
| Port Object<br>-- with one slot in<br>-- message queue. Add<br>-- 12 bytes for each<br>-- additional slot. | 64 | 2 | 96 |
| Surrogate Carrier Object<br>-- with two slots in<br>-- refined carrier. Add<br>-- four bytes for each<br>-- additional slot. | 80 | 3 | 128 |
| Context Object<br>-- does not include space<br>-- in the context for working<br>-- storage or operand stack | 72 | 2 | 104 |
| Domain Object<br>-- only includes fault<br>-- object and trace object<br>-- ADs. Add four bytes for<br>-- each additional AD. | 16 | 1 | 32 |
| Instruction Object<br>-- only includes header,<br>-- not instructions<br>-- themseves. | 24 | 1 | 40 |
| Type Definition Object | 40 | 2 | 72 |
| IP Processor Object | 616 | 5 | 696 |
| IP Process Object<br>-- includes IP context | 536 | 7 | 648 |

This appendix presents timing information for selected iMAX operations. This information is presented for the use of system designers who need to estimate performance for systems using iMAX. All times presume an 8MHz GDP with minimum memory access times. These figures are not applicable to 432/600 systems because of the longer access times in those systems. Times include instruction fetch and operand reference generation times. A range of times is given for each operator.

Note that times are only presented for those iMAX operators that map directly to in-line GDP hardware operators. Performance figures for other iMAX facilities are not yet available.

| CHAPTER/OPERATION | TIME (usec @ 8 MHz) | NOTES |
|---|---|---|
| DEF Basic Definitions | | |
| Inspect_object | 6 - 14 | |
| Idle | 104 - 140 | just includes the time to execute Idle and not the time used to dispatch the next process on the freed-up processor. |
| STO Storage Management | | |
| create a generic segment | 55 - 90 | plus segment clearing time of 1.875 usec (15 cycles) for each 8 bytes in the new segment. The new segment must be less than or equal to 2,048 bytes (or be allocated from already zeroed memory) or the create operation will fault and iMAX software will clear the segment, requiring added time. |
| create a generic refinement | 42 - 73 | |

| CHAPTER/OPERATION | TIME (usec @ 8 MHz) | NOTES |
|---|---|---|
| COM Interprocess Communication | | |
| Send | 97 - 183 | |
| Receive | 96 - 170 | |
| Surrogate_send | 130 - 308 | |
| Surrogate_receive | 112 - 317 | |
| EXT Extended Types | | |
| Retrieve_type_ definition | 14 - 20 | |
| Create_private | 43 - 80 | |
| Create_public | 43 - 80 | |
| Retrieve_public_ representation | 14 - 20 | |
| Retrieve_ representation | 18 - 24 | |

This glossary defines important terms used in this manual. Some are from iMAX, some are from Ada, and some are from the 432 architecture. Within the definitions, references to other terms defined in this glossary are underscored.

Table GLO-1. Terms Defined

| | | |
|---|---|---|
| access | garbage collection | process globals access segment |
| access descriptor | General Data Processor | |
| access environment | generic objects | processor type |
| access rights | global heap SRO | |
| access type | guardian port | read rights |
| AD rights | | refinement |
| any_access | heap SRO | refinement control object |
| asynchronous interface | interconnect | representation rights |
| Attached Processor | interconnect Processor | |
| | | segment |
| base type | level | specification |
| blocked | level check | stack SRO |
| body | lifetime strategy | storage resource object |
| | LIFO | subtype |
| carrier | local heap SRO | synchronous interface |
| central system | | system object |
| compaction | memory type | system type |
| connection | message | |
| constraint | | task |
| constraint_error | normal memory | type |
| context object | | type control object |
| | object | type definition object |
| defining domain | object descriptor | type manager |
| delete rights | object reference | type rights |
| device abstraction | object table | |
| dispatching mix | operand stack | unchecked copy rights |
| domain | | |
| | package | write rights |
| exception | package type | |
| extended type | Peripheral Subsystem | |
| extended-type object | physical storage object | |
| | port | |
| FIFO | pragma | |
| forwarding | print_name | |
| fragmentation | process | |
| frozen memory | process tree | |

access: an Ada value which is either a reference for some other value, or is the value null if no other value is referenced. Each Ada access value is represented by a 432 access descriptor, and each referenced value is a distinct 432 object.

access descriptor (AD): a reference to a 432 object that restricts operations on both the object using the AD (access rights) and on the AD itself (AD rights). The 432 hardware ensures that access descriptors and the objects they refer to can only be manipulated in controlled ways.

access environment: The set of all 432 objects that can be directly or indirectly accessed from a given context. The access environment of a context is determined by its defining domain, by the process globals access segment of the process that contains the context, and by any object references passed as parameters from its callers, returned as results from operations the context calls, received as messages in interprocess communication, or created during the context's execution.

access rights: attributes of a 432 access descriptor (AD) that restrict operations on the referenced object using the AD. Access rights consist of base rights, which restrict the rights to read or write the referenced object, and type rights, which restrict the right to execute certain high-level operations using the AD (e.g., the right to send a message to a port).

access type: an Ada type whose values are references for objects of a second type, specified when the access type is declared. The 432 extensions to Ada specify one predefined access type, any access; its values can be null or references for 432 objects of any other Ada type.

AD rights: attributes of a 432 access descriptor (AD) that restrict operations on the AD itself. AD rights consist of delete rights and unchecked copy rights.

any access: an Ada access type predefined by the 432 extensions to Ada that includes any access value for any type of referenced value. Users must instantiate the Ada generic subprogram Unchecked_conversion to convert from any_access to other access types.

asynchronous interface: an interface to iMAX input/output based on interprocess communication. The requestor of service sends a message describing the desired service to a request port, and receives the message back from a reply port with a reply code indicating either success or some exceptional condition. The interface is called asynchronous because control returns to the requestor before the request is actually processed.

Attached Processor (AP): a processor, usually an Intel microprocessor, which controls the Peripheral Subsystem (PS). The PS contains peripheral devices, controllers, memory, the AP, and a 432 Interface Processor (IP), all communicating on the Peripheral Subsystem's bus. Software running on the AP controls the IP.

base type: 1. a 432 object type field that distinguishes between access segments, containing only access descriptors, and data segments, containing anything but access descriptors. 2. The Ada type from which a particular subtype is derived by applying constraints is called the base type of the subtype.

blocked: the state of a carrier that is queued at a full port waiting to send a message, or is queued at an empty port waiting to receive a message. A process or processor is "blocked" if its carrier is blocked.

body: an Ada unit containing the declarations and statements that implement a subprogram, package, or task specification.

carrier: a 432 system object that carries messages to and from ports, and that may optionally be forwarded to a second port after completing a primary operation. A carrier can be a process carrier, processor carrier, or a surrogate carrier. Process carriers and processor carriers directly represent processes and processors in port operations; if such a carrier must wait for a port operation to complete, then the corresponding process or processor waits as well. However, a surrogate carrier, while it normally acts on behalf of some process, does not cause any process to wait when it must wait, and multiple surrogate carriers can act on behalf of a single process.

central system: the main 432 system in which multiple General Data Processors (GDPs) and Interface Processors (IPs) share a common memory and concurrently execute. I/O and initialization for the central system is provided by one or more Peripheral Subsystems (PSs). The 432 Interface Processors are part of both systems and provide the bridge between them.

compaction: an iMAX memory management service which relocates objects in memory to combine fragmented free storage blocks, thus allowing the allocation of larger segments. Compaction runs concurrently with user programs and its operation is invisible (except in timing). Only normal memory is compacted; frozen memory is not.

connection: an iMAX object type which represents the asynchronous interface for a particular I/O device.

constraint: An Ada restriction on the set of possible values of a type or subtype. A range constraint specifies lower and upper bounds on the values of a scalar type. An accuracy constraint specifies the relative or absolute error bound on values of a real type. An index constraint specifies lower and upper bounds on an array index. A discriminant constraint specifies particular values of the discriminants of a record or private type.

constraint error: a built-in Ada exception raised by iMAX when a run-time constraint is violated. iMAX most commonly raises constraint_ error because an access value passed as a parameter to some iMAX operation lacks needed access rights.

context object: a 432 system object that represents one activation of a subprogram.

defining domain: the domain through which the current context was called. This domain is a major part of the context's access environment. The caller usually possesses an access for just some refinement of the domain (its "public part"), but the called context is able to access all of the domain.

delete rights: an attribute of a 432 access descriptor (AD) that restricts the right to overwrite the AD with a new access value. If an AD is not null and delete rights are absent, then the access value can be eliminated only by reclaiming the segment that contains the AD.

device abstraction: a collection of iMAX I/O functions, defined by a package type, supported by one or more types of devices, and forming a useful abstract device.

dispatching mix: the set of iMAX and user processes that are eligible for dispatching to run on a processor. A process can be idled or blocked at a port and still be "in the mix".

domain: a 432 system object that is a collection of access descriptors that define a major part of the access environment of any context called through the domain. The caller may have access to just a refinement of the domain, called the public part, but the called context can access the entire domain.

exception: An Ada event that suspends normal program execution. Bringing an exception to attention is called raising the exception. An exception handler is program text that responds to the exception.

extended type: A 432 object type defined by software and represented by a 432 type definition object (TDO). Instances of an extended-type appear to users of the type as a two-access record. The first access references the TDO for the type, and the second access references the typed object. An instance of an extended type can have the attribute private which restricts access to the typed object. Instances that are not private are public, and any access to a public instance can be used to obtain an access for the typed object.

extended-type object: an instance of an extended type.

FIFO: First-In-First-Out, a queuing discipline in which the first item to enter a queue is always the first to leave it.

forwarding: the 432 operation of sending a carrier on to a second port after the carrier has been used to send or receive a message at a first port.

fragmentation: the division of free storage into multiple non-contiguous blocks, caused by the normal operation of heap allocation and garbage collection.

frozen memory: a type of memory defined by iMAX within which objects are not swapped out or relocated in physical memory. Frozen memory is not subject to compaction, and fragmentation of frozen storage can cause allocation requests to fail that would otherwise succeed.

garbage collection: a concurrent iMAX process that detects unreferenced objects and reclaims the corresponding descriptors and storage. Garbage collection runs concurrently with user processes but is invisible to them.

General Data Processor (GDP): the main type of processor provided by Intel to execute within the 432 central system. The GDP is a general purpose processor that provides object-oriented addressing and protection, operating system functions in silicon, and hardware floating point arithmetic.

generic objects: A 432 object with a system type (generic) that has no hardware-recognized meaning and can be used to implement arbitrary software structures. Other objects are either system objects or extended-type objects.

global heap SRO: a heap SRO at level zero; only garbage collection can reclaim storage allocated from a global heap.

guardian port: a port to which a process is sent by iMAX when some condition removes it from the dispatching mix.

heap SRO: a 432 storage resource object on which garbage collection is performed to reclaim discarded (unreferenced) objects. Objects can be created and reclaimed in any order using a heap SRO, which can result in fragmentation. Two types of heap SROs are defined, global heap SROs and local heap SROs.

interconnect: a secondary address space used for special-purpose hardware registers associated with initialization, hardware configuration, and hardware error logging. The interconnect address space is organized into special interconnect segments which are normally defined at system initialization.

Interface Processor (IP): a 432 processor that connects a 432 central system to one Peripheral Subsystem (PS). The IP is a slave processor to the Attached Processor (AP) in the Peripheral Subsystem. The IP provides the object addressing and high-level operators needed to access the 432 central system.

level: an ordinal attribute of an object that characterizes the relative lifetime of the object -- a greater level means a shorter lifetime. The level number of a context is always one greater than the level of its caller. The level number is normally 1 for the first context associated with a given process. All objects allocated from a stack SRO have the same level number as the context in which they are allocated, and all are reclaimed when control returns from the context. Objects allocated from heap SROs have the same level number as the heap. Global heap SROs have level number zero, while local heap SROs all have level numbers greater than zero. Objects at level zero can only be reclaimed by garbage collection and never because a context returns. Objects at level zero can only be created from a global heap SRO.

level check: a check when an access descriptor is copied, to ensure that the level number of the destination object is greater than or equal to the level number of the object referenced by the AD. This check ensures that no "dangling references" exist when a context returns and deallocates all objects created in it. The level check is suppressed if the AD being copied has unchecked copy rights.

lifetime strategy: an attribute of objects defined by iMAX that determines when and how an object is deleted, and that derives from the type of SRO used to create the object. The three lifetime strategies are global heap, local heap, and stack.

LIFO: Last-In-First-Out, a dynamic data structure organization in which the last item added to the structure is the first item removed from it.

local heap SRO: a heap SRO that is tied to some context and has a level greater than zero. Objects allocated from a local heap can be reclaimed either by garbage collection or by returning from the associated context.

memory type: an iMAX-defined object and SRO attribute that distinguishes memory that is subject to object relocation (normal memory) from memory that isn't (frozen memory).

message: any global 432 object for which an access descriptor is copied from a sending process to a receiving process.

normal memory: a type of memory defined by iMAX within which objects can be relocated in memory to achieve compaction of free memory, reducing fragmentation.

object: a data structure within memory described by an object descriptor and accessed via access descriptors. Objects are the 432 construct for access control, run-time type checking, storage management, and program addressing.

object descriptor: an object table entry that gives object attributes needed by the 432 hardware, e.g., type and storage information.

object reference: see access descriptor.

object table: a 432 system object containing object descriptors.

object type: type information given in object descriptors for storage segments and refinements, consisting of base type, system type, and processor type.

operand stack: an area within a context data segment that provides an expression evaluation stack for the context.

package: an Ada program unit specifying a collection of related entities such as constants, variables, types, and subprograms. The visible part (public part) of a package contains entities accessible from outside the package. The private part of a package contains structural details hidden from the user of the package; these details complete the specification of the visible entities. The visible and private parts together constitute the package specifications. The package body, which can be separately compiled, contains the bodies (implementations) of subprograms, tasks, or other packages specified in the package specifications.

package type: A set of values characterized by a package specification and including all possible values of the package body. Alternate package bodies are represented by distinct 432 domain objects. Variables can be declared to be of a package type, and alternate package bodies can be dynamically assigned to them at run-time.

Peripheral Subsystem (PS): a computer system controlled by an Attached Processor (AP), which manages one or more peripheral devices and is linked to a 432 central system by a 432 Interface Processor (IP). The PS contains peripheral devices, controllers, memory, the AP, and an IP.

physical storage object (PSO): a 432 system object that provides a free storage pool for use by a 432 storage resource object (SRO).

port: a 432 system object that provides a queuing mechanism supported by hardware with two queues, a bounded message queue, and an unbounded carrier queue. Ports support FIFO and deadline-within-priority queuing. Ports are used for interprocess communication and process scheduling and dispatching.

pragma: An Ada statement that instructs the compiler without changing the meaning of the program unit containing the pragma. For example, pragmas are used for listing control and code optimization control.

print name: a character array type defined by iMAX and used by many iMAX type managers to provide symbolic names. Such names do not securely identify objects, because nothing stops an iMAX user from giving the same print_name to multiple objects.

process: a 432 system object that represents part of a program that can execute concurrently with other parts, also represented as processes. Because processes can compete for execution if there are fewer physical processors than processes, scheduling information is associated with processes for use in selecting the next process to run and to insure that a process does not monopolize a processor for longer than some time limit. A program can consist of one or more processes.

process tree: a group of iMAX processes arranged in a hierarchy of "parent" and "child" processes. Certain process control operations applied to the root process of such a tree effect the entire tree, without the caller needing to know the tree's structure.

process globals access segment (PGAS): a 432 generic access segment that is designated by a process as its global access segment and that provides access to additional attributes of a process's run-time environment. The first few PGAS entries are defined by iMAX and used by the Ada compiler. Several following entries are reserved by Intel; subsequent entries can be safely modified by the user.

processor type: a 432 object type field; each of its values designates the kinds of processors that can reference segments with that processor type. The alternatives now defined are GDP only, IP only, and "all" (both GDP and IP).

read rights: attribute of a 432 access descriptor (AD) that controls the right to read an object.

refinement: a 432 segment that is contained within another segment. When a refinement is created, the displacement of the base of the refinement in the underlying segment and the size of the refinement are specified and checked by the hardware.

refinement control object (RCO): a 432 system object that provides the right to create refinements that are system objects (no special right is needed to create a generic refinement). Each RCO specifies the type of system object that can be created using it.

representation rights: attributes of a 432 access descriptor (AD) that restrict the rights to read or write the referenced object. Representation rights consist of read rights and write rights.

segment: a set of contiguous memory loctions, from 1 to 65,536 bytes in apparent size, defined by a 432 object descriptor. A segment can be in the storage address space, or the interconnect address space, or it can be a refinement of another segment in the storage address space.

specification: an Ada description of the interface provided by a subprogram, package, or task, which can be compiled apart from the body that implements the interface.

stack SRO: a 432 storage resource object built-in to the process object of a process and used for allocation of context objects and objects whose lifetimes are local to the creating context. Allocation and deallocation for a stack SRO are strictly Last-In-First-Out (LIFO). There is no object descriptor for a stack SRO, though it is conceptually a distinct object.

storage resource object (SRO): a 432 system object that provides for the dynamic creation of objects by specifying an object table in which to allocate the object descriptor for a new object, and by specifying a physical storage object (PSO) that specifies a free storage pool from which the new object can be allocated. An SRO specifies the lifetime strategy and memory type of objects allocated from it.

subtype: an Ada data type defined by constraints on the set of values of some other type, called the base type. All built-in and user-defined operations on the base type are inherited by the subtype.

synchronous interface: an interface to iMAX I/O that provides different subprogram interfaces for different I/O services, and that return to (or raise an exception in) their caller only after the requested operation is completed.

system object: a 432 object with a system type that indicates that it has a special role recognized by the hardware. Other objects are either generic objects or extended-type objects.

system type: a 432 object type field that, in conjunction with the base type field, distinguishes a class of 432 objects with a particular hardware-recognized meaning (or lack thereof).

task: in Ada, a process that can execute concurrently with other processes, with certain Ada-specific operations defined on it.

type: in Ada and in general, a set of values with certain operations and representations defined for the set.

type control object (TCO): A 432 system object that provides the right to amplify specific rights on access descriptors for objects of specific types, or the right to create objects of a specified type.

type definition object (TDO): A 432 system object that designates values of a particular software-defined extended type.

type manager: an Ada package or 432 domain object that defines all basic operations on a certain data item or class of data. Any other operations on the item or class must be composed by using the basic operations. A type manager may distribute accesses for the data items, but normally keeps to itself the right to directly read or write the data items.

type rights: attributes of a 432 access descriptor (AD) that restrict the right to execute certain instructions using the AD, depending on the type of system object it references. For example, create rights are required to create a new segment using an SRO access.

unchecked copy rights: an attribute of a 432 access descriptor which, if present, suppresses the level check when the AD is copied. When a new object is created, unchecked copy rights are set on the returned AD only if the new object's level is zero, in which case the level check could never fail. Amplifying unchecked copy rights is a privileged operation that is not available to iMAX users.

write rights: an attribute of a 432 access descriptor that controls the right to write the referenced object.

## REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Inte product users. This form lets you participate directly in the publication process. Your comments will helr 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 o this publication. If you have any comments on the product that this publication describes, please contac 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 foi improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types oi 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 ______

COMPANY NAME/DEPARTMENT ______

ADDRESS ______

CITY ______ STATE ______ (COUNTRY) ZIP CODE ______

Please check here if you require a written reply. ☐

'D LIKE YOUR COMMENTS . . .

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**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**

INTEL CORPORATION, 3585 SW 198th Avenue, Aloha, Oregon 97007 • (503) 681-8080

Printed in U.S.A./Y119/5K/0582/AP