iAPX 432
GENERAL DATA PROCESSOR ARCHITECTURE REFERENCE MANUAL

INTEL432

Copyright © 1981, 1982, 1983, Intel Corporation

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 171860-004

iAPX 432 GENERAL DATA PROCESSOR

ARCHITECTURE REFERENCE MANUAL

Order Number: 171860-004

Copyright © 1981, 1982, 1983 Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

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

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

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

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

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

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

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

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

AEDIT
BITBUS
BXP
COMMputer
CREDIT
i
I^{2}ICE
iATC
ICE
iCS
iDBP
iDIS
iLBX
i$_m$
iMMX
Insite
Int$_e$l
int$_e$l
Int$_e$lBOS
Intelevision
int$_e$ligent Identifier
int$_e$ligent Programming
Intellec
Intellink
iOSP
iPDS
iRMX
iSBC
iSBX
iSDM
iSXM
Library Manager
MCS
Megachassis
MICROMAINFRAME
MULTIBUS
MULTICHANNEL
MULTIMODULE
Plug-A-Bubble
PROMPT
Promware
Ripplemode
RMX/80
RUPI
SYSTEM 2000
UPI

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

MULTIBUS is a patented Intel bus.

Copyright © 1983, Intel Corporation

REV.	REVISION HISTORY	DATE
-001	Original issue Release 1.0 GDP components	01/81
-002	Advance partial issue, revised for Release 2.0 GDP components	10/81
-003	Advance partial issue, revised for Release 3.0 GDP components	10/82
-004	Completed manual for Release 3.2 GDP components	01/84

PREFACE

This manual describes the architecture of Intel's iAPX 432 General Data Processor (GDP). This architecture provides unique support for:

- flexible run-time protection of programs and data
- operating systems and modular programming languages
- multiprocessing
- accurate and robust numerical computation
- reliable and fault-tolerant hardware and software

ARCHITECTURE VS. IMPLEMENTATION

This manual describes the architecture, but not the implementation, of the iAPX 432 GDP. The architecture consists of the processor instruction set, processor-recognized data structures (object set), and actions in response to exceptional conditions (e.g., faults or trace events). The GDP architecture is that information that a compiler writer or systems programmer needs to know about the GDP.

The GDP implementation is all information about the GDP that is not specified by the architecture, such as number of VLSI chips, clock speed, hardware signals, and operator execution times.

A clear separation of architecture and implementation allows the implementation to be changed without changing any software, because the software should depend on only the architecture.

COMPONENT RELEASES

There have been several releases of iAPX 432 GDP components, which have modified the GDP architecture as well as the implementation. This manual describes Release 3.2 GDP components, first released in August 1983. Intel expects, but does not guarantee, that any further releases of iAPX 432 components will be limited to correcting errors, improving implementation, or making upward-compatible extensions to the architecture. That is, further component releases should not require user software modifications.

iAPX 432 PROCESSOR BASE ARCHITECTURE

The iAPX 432 architecture supports multiple types of processors in a single iAPX 432 system, as well as multiple instances of a single processor type. At this writing, one other processor type is available, the iAPX 432 Interface Processor (IP). Both the IP and GDP share a common base architecture that includes object addressing and protection, interprocessor communication, and interprocess communication. This manual defines the common base architecture as part of describing the GDP architecture. The iAPX 432 Interface Processor Architecture Reference Manual relies on this manual to describe shared architectural features.

REFERENCES

The iAPX 432 Interface Processor is described in:

> iAPX 432 Interface Processor Architecture Reference Manual, Order Number 171863.

The iAPX 432 hardware interconnection architecture, used to interconnect processor, memory, and bus subsystems, is described in:

> iAPX 432 Interconnect Architecture Reference Manual, Order Number 172487.
>
> Chapter 18: "A Design Methodology for Highly Reliable Systems: The Intel 432," The Theory and Practice of Reliable System Design, Sieworik and Swartz, Digital Press, 1982.

These data sheets describe the iAPX 432 components:

> iAPX 43201/iAPX 43202 VLSI General Data Processor, Order Number 171873.
>
> iAPX 43203 VLSI Interface Processor, Order Number 171874.
>
> iAPX 43204/iAPX 43205 Fault Tolerant Bus Interface and Memory Control Units, Order Number 210963.
>
> Electrical Specifications for iAPX 43204 Bus Interface Unit (BIU) and iAPX 43205 Memory Control Unit (MCU), Order Number 172867.

Programs for iAPX 432 systems are developed using the Intel 432 Cross Development System, described by these manuals:

> Introduction to the Intel 432 Cross Development System, Order Number 171954.
>
> Intel 432 Cross Development System VAX* Host User's Guide, Order Number 171870.

*VAX is a trademark of Digital Equipment Corporation.

Intel 432 Cross Development System Workstation Reference Manual, Order Number 172097.

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

Asynchronous Communication Link User's Guide, Order Number 172174.

Intel's System 432/600 is a family of microcomputer boards and systems that use iAPX 432 components or support iAPX 432 systems. These manuals describe the System 432/600:

System 432/600 System Reference Manual, Order Number 172098.

System 432/600 Hardware Reference Manual Volume 1, Order Number 172100.

System 432/600 Hardware Reference Manual Volume 2, Order Number 172172.

System 432/670 Installation and Maintenance Manual, Order Number 172101.

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

The Ada* programming language is used to write iAPX 432 programs. The Ada language, features of Intel's implementation of Ada, and Intel's extensions to Ada are described in these manuals:

Reference Manual for the Ada Programming Language, Order Number 171869.

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

NOTE

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

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

iMAX 432, Intel's Multifunction Applications Executive, is the iAPX 432 operating system. iMAX enhances the iAPX 432's unique architectural support for storage management, concurrent processing, and other operating system functions. iMAX 432 is described in:

iMAX 432 Reference Manual, Order Number 172103.

A Programmer's View of the Intel 432 System, Elliott I. Organick, McGraw-Hill, New York, 1983.

MANUAL ORGANIZATION

This manual is divided into two major parts. Part I is a tutorial presentation of the architecture's features and describes how these features interact with each other and with software in a functioning system. Part II gives reference information for the architecture, such as operator descriptions.

Part I contains these chapters:

- Chapter 1, "Introduction," introduces the major concepts of the architecture.
- Chapter 2, "Program Organization," explains the iAPX 432 objects and operators used to represent high-level programming language structures. Domains and instruction objects represent the static structure of a program as a network of modules and subprograms. Processes and contexts represent the dynamic structure of program execution as a hierarchy of tasks and subprogram calls. This chapter also describes the architecture's support for program modules that provide all operations on objects of a particular type. Such type manager modules are the basis for hardware protection of both system- and user-defined object types.
- Chapter 3, "Object Addressing," explains how the iAPX 432 translates local object addresses to virtual object addresses and then to physical addresses.
- Chapter 4, "Memory Management," describes iAPX 432 memory management services and how they are supported by the architecture. These services are provided by a combination of hardware and operating system software.
- Chapter 5, "Parallel Processing," describes the architecture's support for multiple concurrent processes executing in parallel on multiple processors. This chapter describes process communication, process and processor synchronization, and process and processor scheduling and dispatching.
- Chapter 6, "Processor Management," describes GDP caches, interprocessor communication (IPCs), processor dispatching modes, and system initialization.
- Chapter 7, "Instruction Interface," describes the fields of GDP instructions and also all the GDP addressing modes.

- Chapter 8, "Computational Data Types," describes the form and interpretation of the GDP's computational data types. Operations on the types and conversion between types are covered. Floating-point data types and operations are completely described.

Part II contains these chapters:

- Chapter 9, "Object Set," defines the set of processor-recognized data structures, which support operating systems, object addressing, high-level languages, and multiprocessing.
- Chapter 10, "Operator Set," describes all GDP operators. Each operator description specifies the operator encoding, any parameters or results, and the action of the operator.
- Chapter 11, "Instruction Encoding," describes the instruction field encodings for operators, operands, and addressing modes.
- Chapter 12, "Fault and Trace Reference," describes the different faults that can be raised by the GDP, their encodings, and the format of the "fault areas" into which fault information is written. This chapter also describes the GDP's encoding of trace information, used to support software debugging tools.

A glossary follows Part II and defines new terms introduced by the iAPX 432 architecture.

ABBREVIATIONS

The following abbreviations are used in this manual:

AD Access Descriptor
AP Access Part (of an object)
DP Data Part (of an object)
DTO Dynamic Type Object
EDV Embedded Data Value
ENV Entered Access Environment
GDP General Data Processor
IP Interface Processor
IPC Interprocessor Communication
OD Object Descriptor
OT Object Table
OTD Object Table Directory
OTE Object Table Entry
PCO Processor Communication Object
PSO Physical Storage Object
SCO Storage Claim Object
SRO Storage Resource Object
TCO Type Control Object
TDO Type Definition Object

CONTENTS

CHAPTER 1
INTRODUCTION

PAGE

Software Systems 1-1
    Reliability 1-2
    The Semantic Gap 1-2
Objects 1-4
    Object Protection 1-6
    Dynamic Storage Management 1-9
    Object Types 1-10
Programming Systems Support 1-14
Multiprocessing 1-17
Input/Output Architecture 1-19
Computation 1-21
Conclusion 1-23

CHAPTER 2
PROGRAM ORGANIZATION

Procedures 2-1
Packages 2-1
Information Hiding 2-1
Instruction Objects 2-3
Domain Objects 2-4
Static Program Organization 2-5
Context Objects 2-7
    Contexts Vs. Procedues 2-8
    Access Environment 2-8
    Context Description 2-12
    Preallocated Contexts 2-16
    The Call Operators 2-17
    The Return Operators 2-17
    Context Level Numbers 2-17
Process Objects 2-20
Object Managers 2-20
Type Managers 2-20
    Type Manager Implementation 2-22
    Software-Defined Protected Types 2-22
    Creating Typed Objects 2-24
    Type Manager Schema 2-24

CHAPTER 3
OBJECT ADDRESSING

PAGE

Physical Address Spaces 3-1
Two-Part Memory References 3-2
Two-Level Address Mapping 3-2
Address Mapping for Object Protection 3-2
Object Format 3-4
Access Descriptor Format 3-5
Access Selectors 3-6
Access Selector Format 3-6
Address Mapping for Dynamic Storage Management 3-7
Two-Level Object Table Structure 3-8
Overview of Object Addressing 3-9
Address Space Summary 3-9
Refinement Addressing 3-11
Interconnect Addressing 3-12

CHAPTER 4
MEMORY MANAGEMENT

Object Scope 4-2
Objects for Memory Management 4-2
    Storage Resource Objects 4-3
    Object Tables 4-4
    Physical Storage Objects 4-4
    Storage Claim Objects 4-4
Object Creation 4-5
Object Lifetime Strategies 4-6
    Stack Lifetimes 4-6
    Global Heap Lifetimes 4-6
    Local Heap Lifetimes 4-7
Fragmentation and Compaction 4-7
Memory Management Transitions 4-7
Virtual Memory 4-8
Frozen Memory 4-10
Multiple Processors and Memory Management 4-10

CHAPTER 5
PARALLEL PROCESSING

Processes 5-1
Interprocess Communication 5-2
    Messages 5-4
    Ports 5-5
    Carriers 5-5
    Sending Messages 5-6
    Receiving Messages 5-6
    Forwarding Carriers 5-7
Process and Processor Synchronization 5-8
Transparent Multiprocessing 5-9
Process Scheduling 5-10

CHAPTER 6
PROCESSOR MANAGEMENT

PAGE

GDP Caches 6-1
Data Object Cache 6-1
Object Table Cache 6-2
Context Cache 6-3
Process Cache 6-4
Processor Cache 6-4
Cache Summary 6-5
Interprocessor Communication 6-6
Normal GDP Execution Cycle 6-7
GDP Dispatching Modes 6-7
GDP Initialization 6-8

CHAPTER 7
INSTRUCTION INTERFACE

Instruction Execution Environment 7-1
Current Context 7-1
Instruction Objects 7-2
Instruction Stream 7-3
Operand Addressing 7-6
Operand Types 7-6
Operand Alignment 7-7
Logical Address Components 7-7
Operand Addressing Modes 7-9
Branch References 7-22
Large Array Indexing 7-23
Operand Stack Interaction 7-24
Instruction Interpretation 7-26
Physical Address Generation 7-27
Instruction Execution 7-29

CHAPTER 8
COMPUTATIONAL DATA TYPES

Overview of Computational Data Types 8-1
Character Data Type 8-2
Short-Ordinal Data Type 8-2
Ordinal Data Type 8-2
Short-Integer Data Type 8-2
Integer Data Type 8-2
Short-Real Data Type 8-2
Real Data Type 8-2
Temporary-Real Data Type 8-3
Operators for Computational Data Types 8-3
Bit Field Manipulation 8-4
Data Type Conversion 8-5

CHAPTER 8 (CONTINUED)
COMPUTATIONAL DATA TYPES PAGE

GDP Floating-Point Data Types 8-6
General Floating-Point Format 8-9
Classification of Floating-Point Numbers 8-9
Normalized Floating-Point Numbers 8-10
Exponent Biases 8-11
GDP Floating-Point Operand Interpretation 8-12
Floating-Point Rounding 8-21
Data Operator Faulting 8-22
Classification of Data Operator Faults 8-22
Floating-Point Faulting 8-24

PART II REFERENCE INFORMATION

CHAPTER 9
OBJECT SET

Chapter Conventions 9-1
Reserved Fields 9-1
Preserved Fields 9-1
Object Illustration Convention 9-2
Encoded Values 9-2
Index Fields 9-2
Displacement Fields 9-3
Object Representation 9-3
General Storage Segment Structure 9-3
Access Part 9-5
Data Part 9-5
Access Descriptor 9-6
Embedded Data Value 9-8
Object Lock 9-9
Object Descriptions 9-10
System Object Types 9-10
Object Table Object 9-12
Processor Object 9-27
Processor Communication Object 9-33
Process Object 9-35
Context Object 9-43
Domain Object 9-48
Instruction Object 9-49
Port Object 9-51
Carrier Object 9-55
Storage Resource Object 9-59
Storage Claim Object 9-61
Physical Storage Object 9-62
Type Definition Object 9-65
Dynamic Type Object 9-66
Type Control Object 9-67

CHAPTER 10
OPERATOR SET

PAGE

Functional Index of Operators 10-1
Data Operators 10-1
Object Operators 10-6
Operator Descriptions 10-9
Operand Types 10-11
Data Operators 10-15
Character Operators 10-15
Short-Ordinal Operators 10-20
Short-Integer Operators 10-26
Ordinal Operators 10-32
Integer Operators 10-39
Short-Real Operators 10-45
Real Operators 10-51
Temporary Real Operators 10-57
Object Operators 10-63
Sub-Operator Procedures 10-63
Branch Operators 10-65
Access Descriptor Operators 10-68
Type and Rights Manipulation Operators 10-69
Refinement Operators 10-72
Object Creation Operators 10-76
Access Inspection Operators 10-78
Access Interlock Operators 10-80
Context Operators 10-83
Process Communication Operators 10-89
Processor Communication Operators 10-102
Interconnect Operators 10-104
Block Move Operators 10-105

CHAPTER 11
INSTRUCTION ENCODING

Chapter Conventions 11-1
Instruction Fields 11-1
Class Field Encodings 11-2
Format Field Encodings 11-3
Reference Field Format 11-4
Data Reference Formats 11-4
Scalar Data Reference 11-4
Record Item Data Reference 11-5
Static Array Element Data Reference 11-5
Dynamic Array Element Data Reference 11-6
Indirect Reference Field Formats 11-6
Access Selection Field Formats 11-7
Access Selector Formats 11-8
Branch Reference Formats 11-9
Opcode Encoding Summary 11-10

CHAPTER 12
FAULT AND TRACE REFERENCE PAGE

Fault Reference 12-1
Fault Area Formats 12-2
Fault Codes 12-9
Trace Reference 12-35
Trace Operation 12-35
Trace Control Data Area 12-37

GLOSSARY

TABLES

TABLE	TITLE	PAGE
1-1	iAPX 432 System Objects	1-11
1-2	Objects and Functions for Programming Systems Support	1-16
6-1	GDP IPCs	6-6
7-1	Format Field Encodings	7-5
8-1	Significand Sizes	8-11
8-2	Exponent Sizes and Biases	8-11
8-3	Short-Real Operand Classifications	8-13
8-4	Real Operand Classifications	8-15
8-5	Temporary-Real Operand Classifications	8-16
8-6	Signed Zeros	8-20

FIGURES

FIGURE	TITLE	PAGE
1-1	The Semantic Gap in Models of Memory	1-3
1-2	An iAPX 432 Object	1-4
1-3	Threefold Object Protection	1-8
1-4	Refinement Object	1-13
1-5	iMAX 432 Complements the iAPX 432 Architecture	1-14
1-6	Input/Output Architecture	1-20
1-7	iAPX 432 Computational Data Types	1-21
1-8	iAPX 432 Operators and Computational Data Types	1-22
2-1	Instruction Object	2-3
2-2	Domain Object and Refinement	2-5
2-3	Static Program Organization Example	2-6
2-4	Context Object	2-7
2-5	Access Selector	2-9
2-6	Access Environment Example	2-10

FIGURE	TITLE	PAGE
2-7	Context Access Environment	2-11
2-8	Nested Procedures Example	2-14
2-9	Preallocated Contexts Example	2-16
2-10	Dynamic Program Organization Example	2-18
2-11	Object Scopes and Level Numbers in the Dynamic Program Organization Example	2-19
2-12	Type Manager Objects	2-23
3-1	Two-Level Address Mapping	3-3
3-2	Object Format	3-4
3-3	Access Descriptor Format	3-5
3-4	Two-Level Object Table Structure	3-8
3-5	Object Addressing	3-10
3-6	Refinement Object	3-11
4-1	Objects for Memory Management	4-3
4-2	Memory Management Transitions	4-8
5-1	Message AD Transfer	5-4
6-1	GDP Caches	6-5
7-1	Instruction Execution Environment	7-2
7-2	Operand Addressing Overview	7-8
7-3	Base and Index Address Components	7-10
7-4	Data Reference Modes	7-11
7-5	Data Referene Modes	7-11
7-6	Scalar Data Reference	7-12
7-7	Record Item Data Reference	7-13
7-8	Static Array Data Reference	7-14
7-9	Dynamic Array Data Reference	7-16
7-10	Stack Indirect Reference	7-17
7-11	Intrasegment Indirect Reference	7-18
7-12	General Indirect Reference	7-18
7-13	Access Selection Modes	7-19
7-14	Direct Access Selection	7-20
7-15	Stack Indirect Access Selection	7-21
7-16	General Indirect Access Selection	7-22
7-17	Branch References	7-23
7-18	Physical Address Generation	7-27
8-1	Computational Data Types	8-1
8-2	Operators and Data Types	8-3
8-3	Bit Field Operations	8-4
8-4	Data Type Conversions	8-5

CHAPTER 1
INTRODUCTION

"It used to be the program's purpose to instruct our computers; it became the computer's purpose to execute our programs."

--Edsger W. Dijkstra

The iAPX 432 architecture is designed with a single overriding goal: TO IMPROVE SUPPORT FOR SOFTWARE SYSTEMS. This chapter first describes some attributes of software systems and how a computer architecture can aid or hinder the production of quality software. Subsequent sections of this chapter introduce specific features of the iAPX 432 architecture that support software systems.

SOFTWARE SYSTEMS

Modern enterprises and institutions rely on large software systems: management information systems, air traffic control systems, operating systems, etc. Integrated software systems are even larger. An integrated software system may include an operating system, data base manager, text processor, financial modelling software, telecommunications software, and sophisticated graphics. Future integrated software systems will add more capabilities, such as voice and video processing and natural language interfaces. Even small software now depends on the quality of large software, on the translators and operating systems that intervene between almost any program and the computing hardware.

A large software system may contain hundreds of thousands, even millions, of program lines. Such systems are developed by many people working together for several years. During the course of such projects, whole "generations" of computer hardware may come and go, the people working on the project will change, and the requirements of the users or the marketplace will definitely change. To repay the massive development costs, a large software system must have many users over a period of several years. During the years of its use, the system must be "maintained," modified to correct the inevitable errors in such a large product and to adapt to changing hardware and changing user requirements. The activity of maintenance itself frequently introduces new errors; in practice, it has been observed that even mature, well-engineered large software systems contain many errors.

RELIABILITY

Because of their complexity and continuing change, large software systems will contain multiple errors. At the same time, more and more applications of computers demand high reliability of more and more software systems. Obviously, an air traffic control program must be reliable, but so must the compiler and operating system that it depends on. A data base system must be reliable, if it is used by a police department. An engineering design program must be reliable if it is used to design products that could injure people with their flaws. The great costs of finding and correcting errors in released software only increase the importance of software reliability.

Increased support for software reliability is being provided by new "structured" programming languages, by stricter engineering discipline, and also by innovations in computer architecture. Computer architecture can contribute to software reliability in two major ways. The first contribution is to reduce the large gulf ("semantic gap") between high-level programming concepts and the data types and operations provided directly in hardware by a computer architecture. Such a high-level architecture can reduce translator complexity and program complexity, and also improve performance by reducing object code size and providing more functions in hardware. The second contribution is to provide in hardware protection mechanisms that, given that some errors will still exist in large software systems, at least confine errors within a particular module or data set, and prevent errors from propagating to correct modules and their data. The alternative to protection, in which hundreds or thousands of software modules and data objects are mapped into a single large unprotected address space, allows a single erroneous module to corrupt the code or data of any other module in the system's memory. Worse, such errors may be transient and nondeterministic in systems that dynamically load modules or that support multitasking.

Software can be no more reliable than the hardware that executes it, and Intel 432 systems provide comprehensive support for hardware reliability and fault tolerance. This support includes: paired, self-checking VLSI components (including GDPs); ECC (error correcting code) memory; bus parity checking; and support for redundant and reconfigurable buses, memories, and VLSI logic.

THE SEMANTIC GAP

There is a large gulf between the concepts used in modern programming languages and the operations and data types recognized by a particular computer architecture. Computer scientist Peter Denning calls this gulf the "semantic gap." For example, a computer architecture may not provide floating point arithmetic; adding two real numbers on such a computer requires calling a subroutine that may execute hundreds of instructions.

A more important part of the semantic gap is the gap between the models of memory used by programming languages and by computer architectures, as shown in Figure 1-1. A program in a high-level language is a network of program structures and data elements. The network structure itself contains information, e.g., that a particular data element should only be referenced by a particular procedure. Often parts of the network are organized in a hierarchy of nested elements, so that parts of the program are local to another program unit that

contains them. The containing program unit determines the scope of the nested elements; elements that are not nested are global or at "library level." The elements of a high-level language program are also typed; each program element has a fixed type that determines what operations are allowed on the element. For example, an element of type "procedure" can be called but not added to; an element of type "integer" can be added to but not called. In contrast, a conventional computer architecture views memory as a single structureless array of bytes or words. When a translator maps a program into the conventional computer memory, all of the information about network structure, scope, and element types is lost. Also, it becomes expensive (in execution time) to implement varying-length data structures, which often results in constrained and clumsy programs.

Another part of the semantic gap is the presence of features in many computer architectures that have no counterpart in most programming languages. Such features include explicit processor registers, condition codes, and auto-increment/auto-decrement addressing modes. These features make compilation more difficult and increase the temptation to use machine-level code that can take advantage of them. Such features can also increase the complexity of an architecture and actually slow the execution of some operations (such as context switching or recovery from a virtual memory fault).

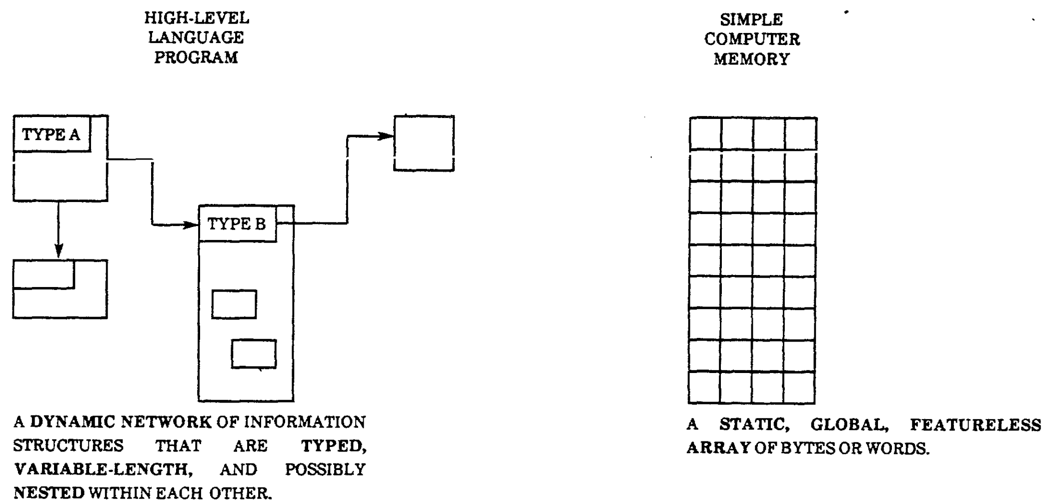


Figure 1-1. The Semantic Gap in Models of Memory

Reducing the semantic gap requires implementing in the hardware architecture the data types, operations, and concepts of modern programming languages. Such improvements in conventional computer architectures include hardware support for floating point computation, subprogram call and return, and array addressing. Conventional architectures still do not support operating system services and structured, typed memory organization.

Reducing the semantic gap has several advantages. First, programs are less complex, because services once provided within the program are now provided by the architecture. Second, performance is improved (faster by more than a hundred-fold for some floating point operations). Third, ad hoc and varying solutions to a software problem are replaced by a single standard mechanism.

OBJECTS

All data structures and program structures in an iAPX 432 system are contained in a network of typed, protected objects. You must understand objects to understand the iAPX 432 architecture. This section describes the reasons for this memory organization, the properties of objects, the different types of iAPX 432 objects, how objects provide protection, and how objects support dynamic storage management.

These are the goals of the iAPX 432 memory structure:

1. Provide memory structures corresponding to the structure of data and programs in modern programming languages.
2. Provide flexible and efficient protection of program modules and data.
3. Support dynamic memory management in which objects are created, deleted, relocated, or "swapped" at run-time.
4. Achieve the first three goals with a unified model, not an ad hoc collection of features.

The resulting structure, called the object model, achieves all these goals and is simple to understand and use. Figure 1-2 shows an iAPX 432 object, its attributes, and access descriptors that reference the object. The chart "Introducing Objects" introduces properties of objects and the object model.

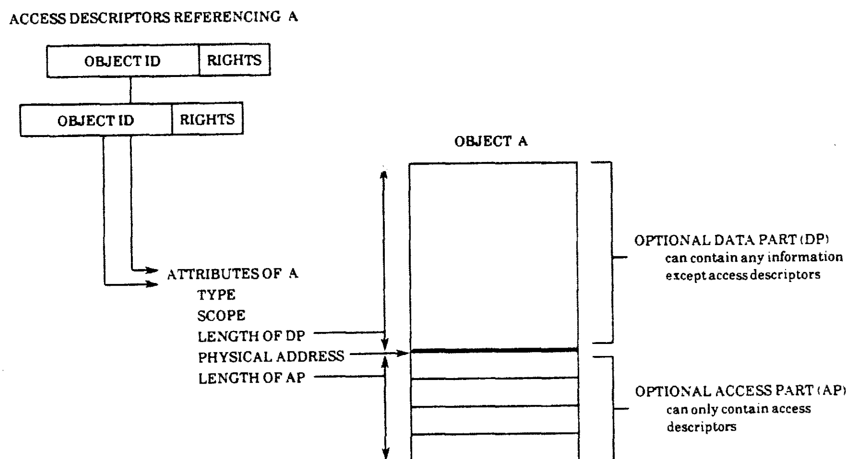


Figure 1-2. An iAPX 432 Object

Introducing Objects

---

1. **All information in an iAPX 432 system is contained in objects.**
   Even the instruction pointer, status flags, and other information used by the GDP are contained in objects.

2. **Each object can have two parts, a data part and an access part.**
   The data part can contain any information except accesses. Data in the data part can be added, assigned, manipulated as bit fields, or used for any purpose other than accessing an object. The access part can contain only access descriptors (ADs). ADs are used for referencing objects and can only be modified in carefully controlled ways.

3. **Objects can be created with different lengths.**
   An object can have from 0 to 65,536 bytes in its data part, and from 0 to 16,384 ADs in its access part. Any reference to any part of an object is automatically checked to ensure that it falls within the bounds of the object.

4. **Each object has a fixed type.**
   The type of an object is determined when the object is created. An object's type can be used to define the operations allowed on the object. Software can define new object types at run-time.

5. **Objects can be local to a program or subprogram call.**
   Each object is created at a particular level that specifies whether the object is global or limited in scope to a particular program or subprogram activation.

6. **Objects can only be read or written via access descriptors.**
   To access data in an object, you must specify an AD that references the object and also specify the offset within the object's data part to the field being accessed.

7. **A procedure call can only access objects for which it has ADs.**
   Each activation of a program or procedure is itself represented by a context object. The instructions executed by the context can only access those objects for which the context has ADs or can obtain ADs.

8. **Access descriptors can provide restricted access to objects.**
   Each AD specifies several rights bits, including read rights and write rights. To read from an object requires read rights set on the AD used; to write to an object requires write rights set on the AD used. Different module activations can have ADs for the same object, but with different rights.

OBJECT PROTECTION

This section describes the protection mechanisms provided by the object model. These protection mechanisms are built into the processor's basic addressing mechanism and are both comprehensive and efficient; special processor structures for parallel checking and for caching frequently-used descriptors contribute to efficiency.

A computer scientist's abstract "protection model" is defined in terms of subjects, the active agents being granted or denied access to information, "objects", the units of information for which access is controlled, and operations, the actions that are individually allowed or disallowed for a particular subject/object combination. In all of these three dimensions of a protection model, the iAPX 432 significantly improves the state of the art. First, the subjects for which access is controlled are not users or large "jobs," but as small as an individual subprogram call. Second, the objects to which access is controlled are not large blocks of contiguous "pages" in some "partition" of a computer's memory, but can be as small as a single one-byte variable (or as large as 128K bytes in a single object). Third, the operations that are allowed or disallowed are not restricted to the primitive read and write operations defined for all objects, but can be extended to include up to three type-specific operations determined by the object's type.

There are other advantages of the iAPX 432 protection model. First, all programs in an iAPX 432 system still share the same virtual address space, allowing pointers to information to be transferred between programs efficiently; this is in contrast with some protection models that "protect" by completely segregating programs in separate virtual address spaces, so that communication requires calls on the operating system and much unnecessary copying of data between the separate address spaces.

A second advantage of the iAPX 432 model is that there is no concept of global privilege. Many architectures have a supervisor mode in which a program can perform any operation and access any system registers or descriptor tables. A program that manages to enter supervisor mode has free run of the system. The supervisor mode is used by many operating system programs, and it is difficult to produce an operating system in which none of those programs can be subverted. In the iAPX 432, there is no supervisor mode, and operating system programs use exactly the same protection mechanism as user programs.

Privileges in the iAPX 432 are type-specific; for each type of object, there is a type manager program module that provides all operations on that type of object. Within a type manager module, there is a type-specific privilege, so that a manager (e.g., for file directory objects) has privileged access to those objects. But the file directory manager could not use its privilege to access another kind of object, such as objects that represent I/O devices. The iAPX 432 architecture supports the principle of least privilege: Each program or subprogram activation has only those privileges that it requires to perform its function. In national security applications, this is called the "need to know" principle, because each activity only has the access it needs to the information it needs.

The chart "Threefold Object Protection" summarizes the three parts of the iAPX 432 object protection mechanism: "need-to-know" access control, extensible object typing, and access rights. Figure 1-3 illustrates how these three parts work together to limit the operations allowed to a particular activity (context).

Threefold Object Protection

I. "Need-to-Know" Access Control

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

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 can allow or prohibit operations unique to an object type (e.g., the right to send a message to a "port" object).
- Type manager software can define new type-specific access rights. For example, the iMAX 432 type manager for processes defines a new access right, called "control rights," for processes.

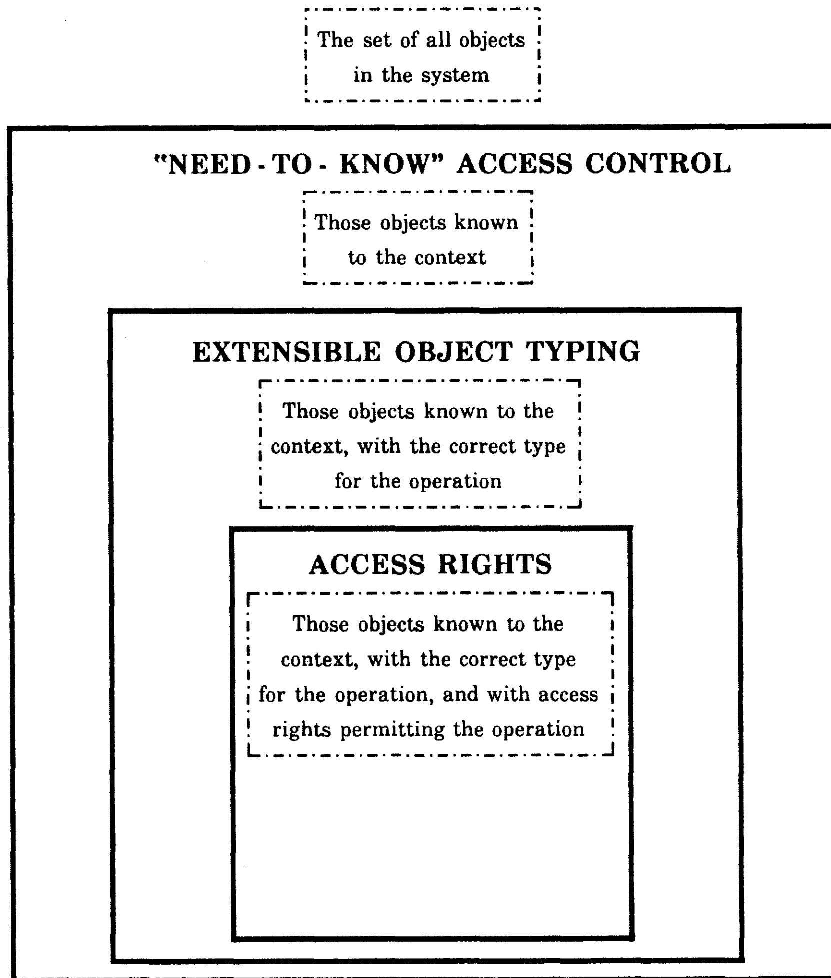


Figure 1-3. Threefold Object Protection

DYNAMIC STORAGE MANAGEMENT

This section briefly describes how the object model supports dynamic storage management:

- Objects can be relocated in physical memory.
- Objects can be swapped in and out of main memory as needed by programs; this feature is called virtual memory.
- Objects can be automatically reclaimed when they are no longer needed, without requiring that a program explicitly delete them.

These services are provided by the architecture and an operating system working together, in cooperation. They are described in more detail in Chapter 4, "Memory Management," and in the iMAX 432 Reference Manual.

Objects can be easily relocated in physical memory because physical addresses for objects are centralized in object descriptors. Each object has one and only one object descriptor. An access descriptor for an object contains an index into a structure of object tables. The object tables contain object descriptors. The object index in an access descriptor selects the object descriptor of the referenced object. This process is described in more detail in Chapter 3, "Object Addressing." An object's descriptor contains its base address in physical memory and also contains several flag bits used by memory management. When an object is relocated in physical memory, only the base address field in its object descriptor must be changed. One of the flag bits can be used to make the object inaccessible while it is being relocated. One reason for relocating objects in memory is to compact memory, so that many small fragments of free memory are combined into one larger free memory block.

To support virtual memory, flag bits are provided that indicate whether or not an object is currently allocated in primary memory, and whether an object has been accessed or altered recently.

To support automatic reclamation of objects when they are no longer needed, a level number is associated with each object when the object is created. The level number indicates the object's scope, i.e., the program or subprogram activation that the object is local to. When control returns from a particular subprogram call, all objects in the program that are local to that call can be deallocated. The deallocation is either done by hardware or by the operating system assisted by hardware, depending on how the objects were allocated. Level numbers are also used to constrain the copying of access descriptors; a level check ensures that an AD for an object is never copied into an object with a longer lifetime (i.e., an AD for an object is never copied outside of the object's scope). Copying an AD for an object outside of the object's scope would be a protection violation; the object could be deleted on exit from its scope, but ADs for the deleted object would still exist that could later be used to reference a different object when the freed object descriptor was reused.

Global objects that are not local to any module activation can only be reclaimed when there are no access descriptors for them. Because an object can only be accessed via an AD, an object with no ADs referencing it is unusable and can be deleted. This process is called garbage collection, and requires an exhaustive search of all objects in memory that could contain access descriptors for the objects that are candidates for deletion. In other systems that offer garbage collection, all other activity in the system must stop whenever garbage collection is run to produce more free memory; other activity may be halted for seconds or even minutes, unacceptable in many computer applications. The iAPX 432 supports parallel garbage collection; the garbage collector executes as one process in a multiprocessing environment, and other system and user processes can run concurrently with garbage collection. The GDP performs one crucial part of garbage collection, setting a flag bit in an object's descriptor whenever an AD for the object is copied. The rest of the algorithm is implemented by the iMAX 432 operating system.

OBJECT TYPES

This section describes the different types of objects recognized by the GDP. These include:

- generic objects
- system objects
- dynamic-type objects
- refinements of any of these objects
- interconnect objects

Generic objects have no processor-recognized meaning. Such objects can be used for any purpose by software. Creating generic objects is faster than creating other types of objects and requires no special privilege. (This makes generic objects unsuitable for being managed by a type manager module, as such a module must control the creation of objects of the managed type.) In a language such as Ada or Pascal, executing a new operation that creates a record referenced by a pointer would, on the iAPX 432, create a generic object and an AD referencing it.

System objects have specific uses and specific formats recognized by the GDP. They are the backbone of the architecture and of the operating system. Much of your effort in reading this book will be in understanding the different system objects. These objects are the major part of the iAPX 432's object set, as much a part of this architecture as the instruction set or register set of another computer. The definition of system objects as part of the architecture makes it possible to place important parts of the operating system into the hardware architecture and to provide high-level services, such as object creation and intertask communication, as single GDP operators. Table 1-1 lists the 14 iAPX 432 system objects.

Dynamic-type objects are objects with a software-defined dynamic type. For example, software could define a distinct type of objects to represent I/O devices in the system. Chapter 2, "Program Organization," describes how dynamic types can be used to protect objects using type manager modules. These protection mechanisms are the same for both system objects and dynamic-type objects. Creating a dynamic-type object (or a system object) is a privileged operation that can be restricted to a type manager module.

Table 1-1. iAPX 432 System Objects

Instruction Object
contains GDP instructions; the GDP will fetch instructions only from instruction objects.

Domain
represents a program module (package) and references subprograms (instruction objects) and data objects in the module.

Context
represents a program or subprogram activation (call) and defines the access environment of the call, i.e., the set of objects that the activation can reference.

Type Definition Object (TDO)
represents a software-defined object type, and can contain attributes of the type (e.g., the type name).

Type Control Object (TCO)
represents type-specific privileges, such as the right to create objects of a particular type or to gain access to objects of a particular type.

Object Table
contains the object descriptors used in object addressing and memory management.

Storage Resource Object (SRO)
represents a free storage pool used to create new objects; references an object table that will contain the new object's descriptor, a physical storage object from which the new segment will be allocated, and a storage claim object that limits allocation from this SRO.

Physical Storage Object (PSO)
specifies free storage blocks in memory.

Storage Claim Object (SCO)
limits the number of bytes that can be allocated from a set of SROs that reference this SCO.

Process
represents a program or subprogram activation that can execute concurrently (in parallel) with other processes.

Port
provides communication between concurrent activities. A port includes a queue of messages sent to the port but not yet received, and a queue of blocked activities waiting to receive messages (at an empty port) or to send messages (at a port with a full message queue).

Carrier
represents an activity in communication with other concurrent activities via ports. Carriers carry messages to and from ports.

Processor Object
contains attributes and state information for an iAPX 432 processor (e.g., a GDP). Because programs in an iAPX 432 system can only manipulate information in objects, all information about a processor that must be visible to software must be contained in an object.

Processor Communication Object
used by the iAPX 432 interprocessor communication mechanism to transfer messages between processors.

Refinements

A refinement is an object that is part of another object (see Figure 1-4). 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 employment record, a refinement of the record can be created. Sensitive information such as salary can be excluded from the refinement and cannot be accessed using the refinement.

Operands in a refinement are addressed exactly as in a simple object, with a displacement from the start of the object (the start of the refinement). A refinement can itself be refined, and there can be several different refinements of one underlying object.

Interconnect Objects

An interconnect object is a special kind of data object allocated in a special interconnect address space. This address space contains hardware registers used for system initialization, interprocessor communication, hardware error reporting, and configuration information. Interconnect objects cannot contain access descriptors.

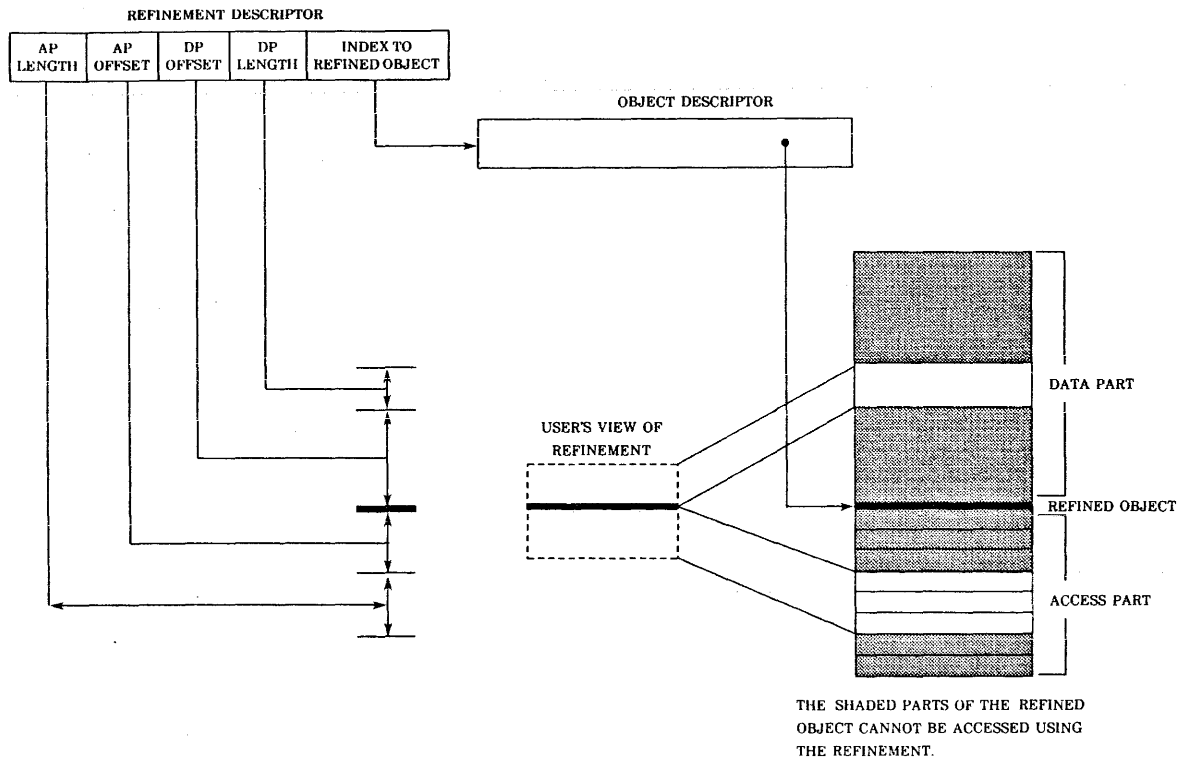


Figure 1-4. Refinement Object

Type information for objects is contained in their object descriptors. The entry type field distinguishes between three types of object descriptors: storage descriptors for normal objects, refinement descriptors for refinements, and interconnect descriptors for interconnect objects. In storage descriptors and refinement descriptors, an object type field provides further type information, consisting of system type and processor type. The system type specifies whether an object is a generic object, dynamic-type object, or one of the 14 types of system objects. The processor type field specifies what types of iAPX 432 processors can reference the object: GDPs only, Interface Processors only, or all processors. Finally, dynamic-type objects and system objects can specify a software-defined dynamic type for the object, which is represented by an AD for a type definition object.

PROGRAMMING SYSTEMS SUPPORT

This section describes how the GDP supports programming systems (e.g., operating systems and compilers). 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 iAPX 432 architecture are called the "Silicon Operating System." These features are not in themselves a complete operating system, but are essential parts of one.

The iAPX 432 functions as a hardware/software partnership. Operations are provided in the hardware for any one 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 the iMAX 432 operating system, cooperating with hardware to provide complete system services (see Figure 1-5).

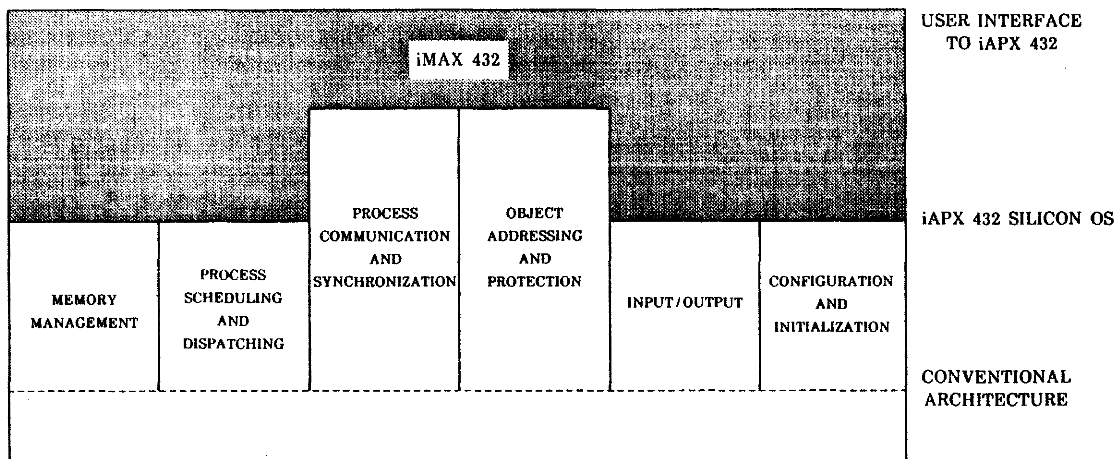


Figure 1-5. iMAX 432 Complements the iAPX 432 Architecture

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

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

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

The iAPX 432 architecture provides major support for programming systems in these areas:

- program organization
- memory management
- parallel processing

Table 1-2 summarizes the objects and functions provided by the GDP in each of these areas. For more information, refer respectively to Chapter 2, "Program Organization," Chapter 4, "Memory Management," and Chapter 5, "Parallel Processing."

Table 1-2. Objects and Functions for Programming Systems Support

PROGRAM ORGANIZATION SUPPORT

Objects	Functions
Instruction Object	CALL/RETURN
Domain	ENTER ENVIRONMENT (change access env.)
Context	COPY PROCESS GLOBALS
Process	RETRIEVE TYPE DEFINITION
Type Definition Object	RESTRICT RIGHTS
Type Control Object	AMPLIFY RIGHTS

MEMORY MANAGEMENT SUPPORT

Objects	Functions
Storage Resource Object	CREATE OBJECT/REFINEMENT
Object Table	CREATE TYPED OBJECT/REFINEMENT
Physical Storage Object	enforce storage claim
Storage Claim Object	clear new segments
	deallocate stack objects on RETURN
	support garbage collection
	support segment relocation
	support virtual memory

PARALLEL PROCESSING SUPPORT

Objects	Functions
Process	SEND/RECEIVE messages at ports
Port	forward carriers to second ports
Carrier	DELAY PROCESS
Processor Object	SEND TO PROCESSOR
Processor Communication Object	schedule processes
	dispatch processes and processors
	LOCK OBJECT/UNLOCK OBJECT
	INDIVISIBLY ADD/INDIVISIBLY INSERT
	service timers
	handle interprocessor messages

Note: iAPX instruction set operator names are capitalized.

MULTIPROCESSING

It is commonplace for a single computer system to handle many activities simultaneously (e.g., multiple terminal users in an office system or multiple sensors and machines in a factory control computer). The different activities that seem to occur in parallel (concurrently) are called processes. The machine that executes the activities is called a processor. In many computer systems that handle multiple processes, execution of the different processes is interleaved on a single processor. This is feasible because each process may spend most of its time waiting for a slower I/O device or may require only a periodic small "slice" of the processor's time to execute at an acceptable speed. The single processor approach to handling multiple processes is cheap and relatively simple, but can have these disadvantages:

1. Such systems often need to offer a range of performance, so that more users and more devices can be easily accommodated. But a particular processor offers a fixed level of performance.

2. Even with advanced and more expensive designs, there is an upper limit to the performance of a single processor, and that limit is inadequate for some applications.

3. If the single processor fails, the entire system fails.

Computer architects are designing systems with multiple processors both to overcome these disadvantages, and because the low cost of microprocessors makes multiple processor systems more economically feasible. Three different ways to use multiple processors have emerged: using specialized processors, distributed systems, and "tightly-coupled" multiprocessors such as the iAPX 432.

Specialized processors can be designed to offload major parts of a main processor's workload. This approach has been quite successful in offloading input/output operations to I/O "channels," and large array calculations to specialized array processors. Other examples of specialized processors are often found in device controllers, such as graphics processors and intelligent disk controllers. Specialized processors can increase system complexity and cost but provide major increases in performance, resulting in a better ratio of price to performance. However, the system that includes specialized processors still suffers from all three disadvantages listed above, albeit at a higher level of performance. (Though the failure of a single specialized processor may only halt some and not all processes in the system.)

Distributed systems replace central computer systems with a network of workstations, each containing a single processor. There is essentially one processor per user, which needs to execute only a few parallel processes (one or a few user tasks and perhaps some simultaneous I/O tasks). Distributed systems overcome all the disadvantages listed above. A system is expanded to handle more users by adding more workstations. The limited performance of a single processor is not a significant constraint because each user has a dedicated processor. If one workstation fails, the user can simply use another workstation. However, distributed systems have some disadvantages of their own:

1. Many computer applications involve access by multiple users and programs to central data bases. The central data bases cannot (with the presently implemented state of the art) be distributed; there are major problems in data base integrity, security, and access in spreading an organization's master files over hundreds of workstations.

2. Each workstation normally requires disk drives and significant local memory, as well as a more expensive processor than is needed by a terminal. To purchase two hundred floppy disk drives for a hundred workstations can cost much more than purchasing the equivalent amount of shared disk storage in a central system.

3. Even with networking software, it is more difficult to share data and programs, to update software, and to do user accounting in a distributed system.

Note that if a data base in a distributed system is centralized at a single node, then all the problems of the single processor system can reappear.

A tightly-coupled multiprocessor contains a number of homogeneous processors on a common bus, executing processes in a shared memory. Applications software can be structured in the same way in a tightly-coupled multiprocessor as in a classical single processor: as a collection of cooperating processes executing within a single computer system. In the multiprocessor there are multiple processors to service the "ready list" of processes ready to run, speeding up throughput. All three of the disadvantages of a single processor are overcome by the tightly-coupled multiprocessor:

- Performance can be increased in increments by plugging in more processors (without changing software).

- System performance can be increased almost without limit by adding processors (and using multiple buses between processors and memory to overcome bus bandwidth limitations).

- If a processor fails, it can be taken out of service and the system can continue to operate with reduced performance using the remaining processors.

The iAPX 432 is a tightly-coupled multiprocessor. Because the number of GDPs can vary without changing software, the iAPX 432 is said to provide transparent multiprocessing. The iAPX 432 can also take advantage of the other multiprocessing approaches where appropriate. Specialized processors are used for input/output in an iAPX 432 system, and other specialized processors can be provided in peripheral subsystems (described below). The small size and cost of the iAPX 432 MICROMAINFRAME™ makes it usable in workstations that can be networked in a distributed system, the other type of multiprocessing system described above. Chapter 5, "Parallel Processing," describes in detail how the iAPX 432 supports multiprocessing.

INPUT/OUTPUT ARCHITECTURE

A major task in computing systems is to quickly and reliably transfer data to and from peripheral devices such as disks, terminals, and printers. Many I/O operations require:

- fast response time to interrupts from devices requesting service
- high throughput for simple data transfer, data conversion, logical, and arithmetic operations

Note that I/O operations do not require the floating point arithmetic or high-level operating system services of the GDP.

I/O can consume more than half the processing time of a general-purpose computer system, a large enough fraction to make special optimization of the I/O function attractive.

The very features that make the GDP powerful for data processing make it poor for I/O:

- The GDP's high-level instructions to provide system services in hardware can consume dozens or even hundreds of microseconds in a single instruction, too great a duration for acceptable interrupt latency.
- The GDP is designed to execute processes without preemption, until they block waiting for some event or until a time slice for the process expires. The GDP caches much information on-chip to speed up process execution. The size of this on-chip information would slow down any attempt to preempt a GDP and switch it to an interrupting activity.
- In a system with multiple GDPs, selection of a processor to interrupt would cause additional overhead.

In the iAPX 432 architecture, there is a division of labor between the GDPs, which provide extensive computation, protection, and support for programming systems, and peripheral subsystems that provide input/output and system initialization services. The peripheral subsystems act as I/O "channels" in an iAPX 432 system. Each peripheral subsystem has a separate bus and address space. Attached to this bus and addressed in the separate address space are one or more I/O devices. A peripheral subsystem also contains memory and at least one attached processor (AP) which provides processing power in the subsystem. An iAPX 432 interface processor (IP) is the bridge between each peripheral subsystem and the iAPX 432 central system. Figure 1-6 illustrates this I/O architecture.

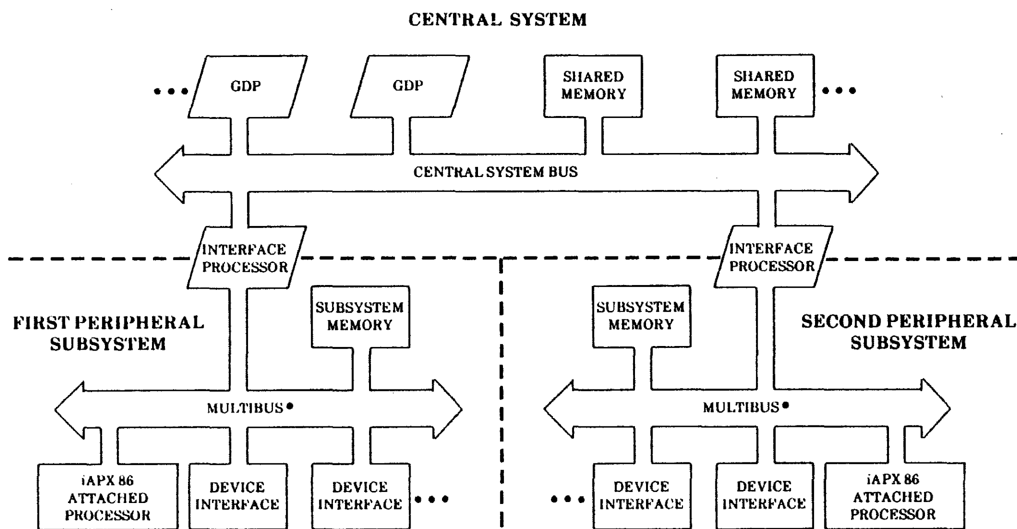


Figure 1-6. Input/Output Architecture

Data is transferred between the central systems and the peripheral subsystems via IP windows that map a portion of the peripheral subsystem address space to the data part of a central system object. Each IP can drive I/O transfers through up to four windows simultaneously (with a fifth window dedicated to controlling the IP). The IP can execute operations on objects in the central system, to send and receive messages that represent I/O transactions for example, or to enter objects into its access environment. When functioning in the central system, the IP uses the same object addressing and protection mechanisms as GDPs. All these IP operations, however, are done at the command of peripheral subsystem software executing on the Attached Processor. To the AP, the IP appears as a slave device, to which the AP writes command codes and operands. Within the central system, the IP, carrying out those commands, functions as an object-based processor. The AP and IP together can be considered a virtual "I/O processor," a partnership in which the AP fetches instructions and provides data addressing, computational operators, and branching operators within the peripheral subsystem, while the IP provides the window mechanism for data transfer and provides object operators within the central system. The I/O architecture is described in much greater detail in the iAPX 432 Interface Processor Reference Manual and the iMAX 432 Reference Manual.

COMPUTATION

The iAPX 432 GDP is a powerful computing engine in addition to its support for modern programming systems. The GDP supports eight computational data types (Figure 1-7) ranging from 1-byte characters to 10-byte extended precision floating point numbers. The GDP supports the proposed IEEE standard for binary floating point arithmetic, the same standard supported by Intel's 8087 numerical coprocessor for the iAPX 86 architecture. This industry standard guarantees significant compatibility and portability of numeric software to and from the iAPX 432. The iAPX 432 also provides distinct signed integer and unsigned integer (ordinal) data types, with overflow checking provided as part of all integer arithmetic operations. These data types and operations are a significant improvement over many processors that require extra instructions to test or manipulate different flag bits in order to do overflow checking or provide both signed and unsigned integer arithmetic. Thirty-two-bit ordinal and integer types are provided by the GDP, in contrast to many processors that must implement such types in software.

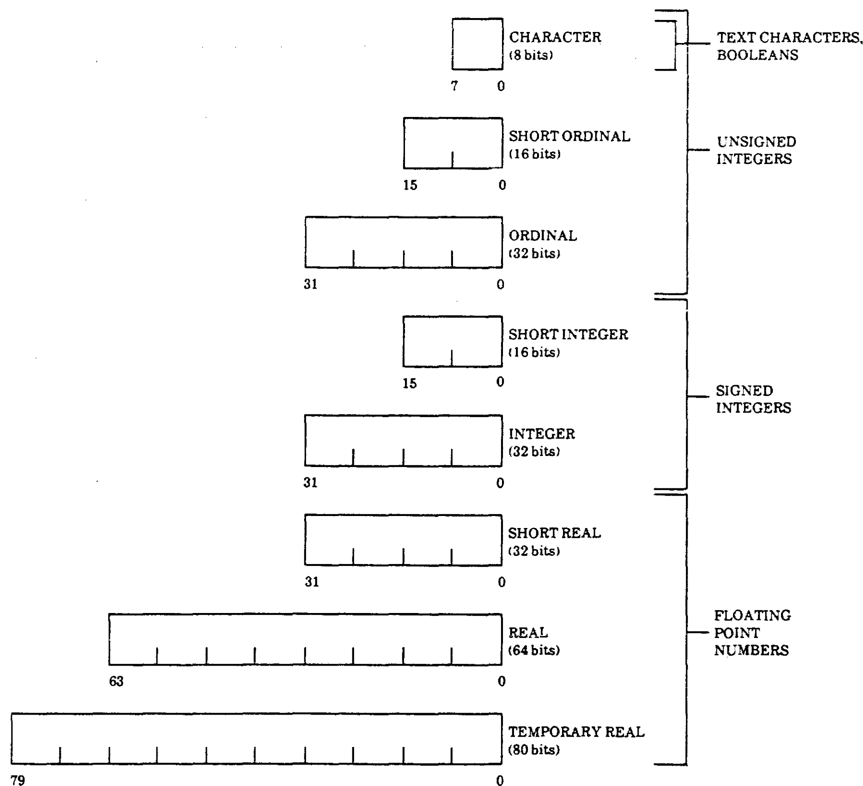


Figure 1-7. iAPX 432 GDP Computational Data Types

Operations provided for these data types include arithmetic, logical, relational, bit-field, and type conversion operators. Table 1-5 shows which operators are available for which computational data types. Chapter 8, "Computational Data Types," describes these types and the operations on them in more detail.

DATA TYPES

	OPERATORS	CHARACTER	SHORT ORDINAL	ORDINAL	SHORT INTEGER	INTEGER	SHORT REAL	REAL	TEMPORARY REAL
MOVE OPERATORS	MOVE	X	X	X	X	X	X	X	X
	SAVE	X	X	X	X	X	X	X	X
	ZERO	X	X	X	X	X	X	X	X
	ONE	X	X	X	X	X	-	-	-
LOGICAL OPERATORS	AND	X	X	X	-	-	-	-	-
	INCLUSIVE OR	X	X	X	-	-	-	-	-
	EXCLUSIVE OR	X	X	X	-	-	-	-	-
	EQUIVALENCE	X	X	X	-	-	-	-	-
	NOT	X	X	X	-	-	-	-	-
ARITHMETIC OPERATORS	ADD	X	X	X	X	X	*	*	X
	SUBTRACT	X	X	X	X	X	*	*	X
	MULTIPLY		X	X	X	X	*	*	X
	DIVIDE		X	X	X	X	*	*	X
	REMAINDER		X	X	X	X	-	-	X
	INCREMENT	X	X	X	X	X	-	-	-
	DECREMENT	X	X	X	X	X	-	-	-
	NEGATE	-	-	-	X	X	X	X	X
	ABSOLUTE VALUE	-	-	-			X	X	X
	SQUARE ROOT								X
	INDEX	-	-	X	-		-	-	-
BIT-FIELD OPERATORS	EXTRACT		X	X	-	-	-	-	-
	INSERT		X	X	-	-	-	-	-
	SIGNIFICANT BIT		X	X	-	-	-	-	-
RELATIONAL OPERATORS	EQUAL	X	X	X	X	X	X	X	X
	NOT EQUAL	X	X	X	X	X			
	EQUAL ZERO	X	X	X	X	X	X	X	X
	NOT EQUAL ZERO	X	X	X	X	X			
	LESS THAN	X	X	X	X	X	X	X	X
	LESS THAN OR EQUAL	X	X	X	X	X	X	X	X
	POSITIVE	-	-	-	X	X	X	X	X
	NEGATIVE	-	-	-	X	X	X	X	X
	MOVE IN RANGE				X	X			
CONVERSION OPERATORS	TO CHARACTER	-				X			
	TO SHORT ORDINAL	X	-			X			
	TO ORDINAL			-		X			X
	TO SHORT INTEGER				-	X			
	TO INTEGER	X	X	X	X	-			X
	TO SHORT REAL						-		X
	TO REAL							-	X
	TO TEMPORARY REAL		X	X	X	X	X	X	-

WHERE:
X MEANS THE OPERATOR IS AVAILABLE FOR THE GIVEN DATA TYPE.
* MEANS THE OPERATOR IS AVAILABLE FOR THE GIVEN DATA TYPE AND FOR INSTRUCTIONS IN WHICH ONE OF THE OPERANDS IS A TEMPORARY REAL.
- MEANS THE OPERATOR IS NOT AVAILABLE AND WOULD BE OF LITTLE OR NO USE IF IT WERE.
(BLANK) MEANS THE OPERATOR IS NOT AVAILABLE.

F-0273

Figure 1-8. iAPX 432 Operators and Computational Data Types

CONCLUSION

This chapter began with the iAPX 432 architecture's overriding goal of improving support for large software systems. It described the importance of large software systems and their reliability, and of detecting and confining the inevitable errors in such systems. The semantic gap between conventional computer architectures and high-level programming concepts was identified as a significant cause of complexity and unreliability in software, especially the gap in models of memory. The object model was introduced as a unifying paradigm that resolves that semantic gap, provides flexible and efficient protection, and also supports dynamic memory management.

The system objects of the iAPX 432 were described, as well as the crucial role they play in the "Silicon Operating System" that places crucial functions into hardware, while maintaining flexibility by carefully separating policy and mechanism.

The importance of multiprocessing was explained, and the iAPX 432's tightly-coupled multiprocessor architecture was compared with alternative designs. The iAPX 432 I/O architecture was introduced as an extension of the basic multiprocessor architecture.

Last, the iAPX 432 GDP's powerful and standard computational capabilities were described, including built-in floating point operations to support the proposed IEEE standard, and also 32-bit integer and ordinal arithmetic.

CHAPTER 2
PROGRAM ORGANIZATION

This chapter describes the iAPX 432 system objects and operators that support high-level language concepts such as procedures and packages. The high-level language concept of a protected "type manager" module is explained in detail, as well as how the architecture provides run-time protection for such modules.

PROCEDURES

A computer program is frequently modularized -- organized as a collection of smaller and more manageable pieces -- as a collection of procedures. Procedures are also called subroutines or subprograms. Each procedure is a sequence of instructions to perform some service. For example, a square root procedure takes a real number, computes the square root, and returns the root to its caller. A procedure is normally called from some other procedure in the program. The calling procedure may pass arguments or parameters that control the action of the called procedure; the called procedure may return result values to its caller.

PACKAGES

A computer program can also be organized as a collection of packages. A package is a collection of related subprograms and data. A package has a two-part definition. The package specification defines the subprogram interfaces and data declarations (types, variables, or constants) available to other parts of the program. The package body defines the implementation of the specified subprograms. The package body can also include subprograms, data declarations, and accesses to other packages that are all hidden from other parts of the program.

INFORMATION HIDING

Information hiding is an important concept in understanding both forms of modularization and in understanding the advantages of organizing a program as a collection of packages. A procedure interface specification or a package specification can be thought of as a contract between the module's developer and the module's "users." (Note that these users are actually other program modules.) The specification describes the form and content of the services the module is to deliver to outside users. Within the requirements set by the specification, the module's implementation can vary: to improve performance, correct errors, or adapt to new hardware or operating systems.

One measure of how well a program is modularized is: "How much information is hidden by the modularization?" For example, a standard I/O interface, like those provided by the programming languages Ada, C, and Modula-2, can successfully hide all details of device registers, interrupt handling, and even hide the operating system. An example of poor modularization is the program with an "initialization module" that is nothing but a sequence of assignment statements that bind values to global variables. The global variables and their structure are still known in the rest of the program, and it is probable that many of the initial values are also known and relied upon in the rest of the program. Such modularization might make a program more readable, but it hides little information and does not make the program more modifiable.

Modularization using procedures can hide complex algorithms easily. For example, a procedure to compute the sine of an angle can be implemented using a Taylor series or using Chebyshev polynomials, without changing the interface to the user. Procedures can also hide complex data structures that arise in the intermediate stages of an algorithm and are local to the algorithm and the procedure.

Procedures cannot hide complex data structures that are not local to any procedure, such as the structure of a disk directory, or an operating system task control block, or the indexing structures for a data base system. In a language like Pascal, any data that must be shared by procedures must be global -- accessible to the entire program.

Packages provide a more powerful and complete form of information hiding. Complex data structures, such as those for implementing a data base, are hidden in the package body. The package specification provides a simple interface with multiple procedures for such operations as Read, Write, Insert, Delete, etc. Such data structures are "global" in the sense that they may exist throughout execution of the program, but "local" to a particular package that hides their representation.

Information hiding using packages has several advantages:

1. Program testing is easier. Most data structures are encapsulated in a package, and their correctness can be assured by reviewing and testing just the package body.

2. Program modification is easier. Not just algorithms but major program data structures can be changed within one module without changing other modules.

3. Programs are more understandable. Packages can be understood in isolation, so long as the specifications of any other packages used are understood. Programs can be organized so that each package manages a single data type or data object.

Increased understanding of how to modularize programs and of what programming language constructs are needed has produced a major improvement in programming methods. The iAPX 432 is the first computer architecture to support these new methods in hardware, as described in the following sections.

INSTRUCTION OBJECTS

The GDP represents a procedure as one or more instruction objects (see Figure 2-1). When a procedure (an instruction object) is called, the GDP requires certain information for the context object that represents the call. This information is contained in an Instruction Object Header in the first eight bytes of the instruction object's data part. The remainder of the data part can contain instructions.

GDP instructions are not aligned on byte or word boundaries and are varying-length sequences of bits. Instruction fields are frequency encoded (Huffman encoded) so that more frequent operation codes, formats, classes, and addressing modes are encoded in fewer bits.

GDP instruction pointers and branch destinations are always bit displacements into the data part of an instruction object. For example, an instruction pointer value of 93 references bit 5 in byte 11 of an instruction object's data part.

Because instruction pointers and branch destinations are computed as 16-bit short ordinals, the maximum instruction bit displacement is 65,535. Thus instruction objects normally will have a data part length less than or equal to 8,192 bytes (65,536 bits).

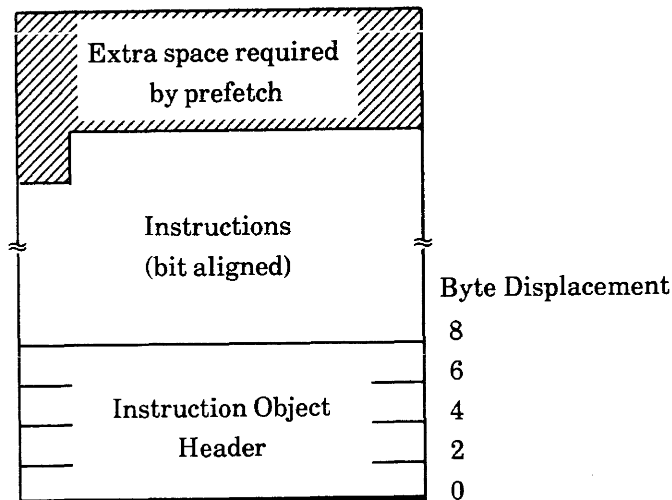


Figure 2-1. Instruction Object

The instruction object data part length should be rounded up from the end of the last instruction to a 16-bit boundary (double-byte boundary) plus 32 bits (four bytes). This extra space is required because the GDP fetchs 32 bits at a time from the instruction stream, aligned on a 16-bit boundary. If the extra space is not provided, the GDP's prefetching from the instruction stream can cause an erroneous fault by attempting to read from beyond the bounds of the instruction object.

The first instruction in a procedure is fetched from bit displacement 64, immediately following the eight bytes reserved for the instruction object header.

The fields of GDP instructions and the different operators and addressing modes are described in Chapter 7, "Instruction Interface," Chapter 10, "Operator Set," and Chapter 11, "Instruction Encoding."

DOMAIN OBJECTS

A domain object represents a package. A refinement of the domain is made available to other packages (other domains) and represents the package specification, the interface that is available to other packages. The portion of the domain that is not contained in the refinement contains or references the information hidden in the package body, which is not available to other packages. Each domain contains two processor-recognized fields, the first two access descriptors in the domain access part. These two ADs reference the fault and trace instruction objects for the domain. If any operation within the domain causes a fault or a trace event, control is transferred to the fault instruction object or the trace instruction object referenced by the domain. Chapter 12, "Fault and Trace Reference," provides more information about these objects and about fault-handling and tracing. Note that the two processor-recognized ADs are normally not included in the domain refinement.

The domain can contain ADs for instruction objects that represent procedures of the package, ADs for objects that contain constants used by those procedures, and ADs for data objects defined by the package. The domain data part can also contain data items defined by the package.

The public part of a domain is that part in a refinement used by other packages. The private part of a domain is that part that is not visible to other packages. Figure 2-2 illustrates a domain and domain refinement.

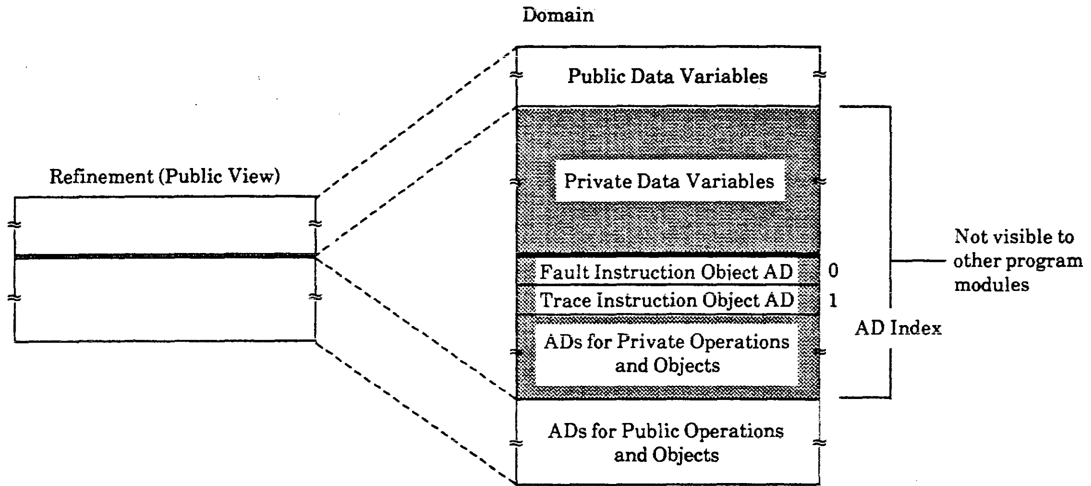


Figure 2-2. Domain Object and Refinement

STATIC PROGRAM ORGANIZATION

The static structure of an iAPX 432 program is represented by a hierarchy of domains and instruction objects, illustrated in Figure 2-3. Each domain provides a different access environment for procedures within it. As control flows from one domain to another, the set of objects that can be referenced changes. This process is described in more detail in the following description of context objects.

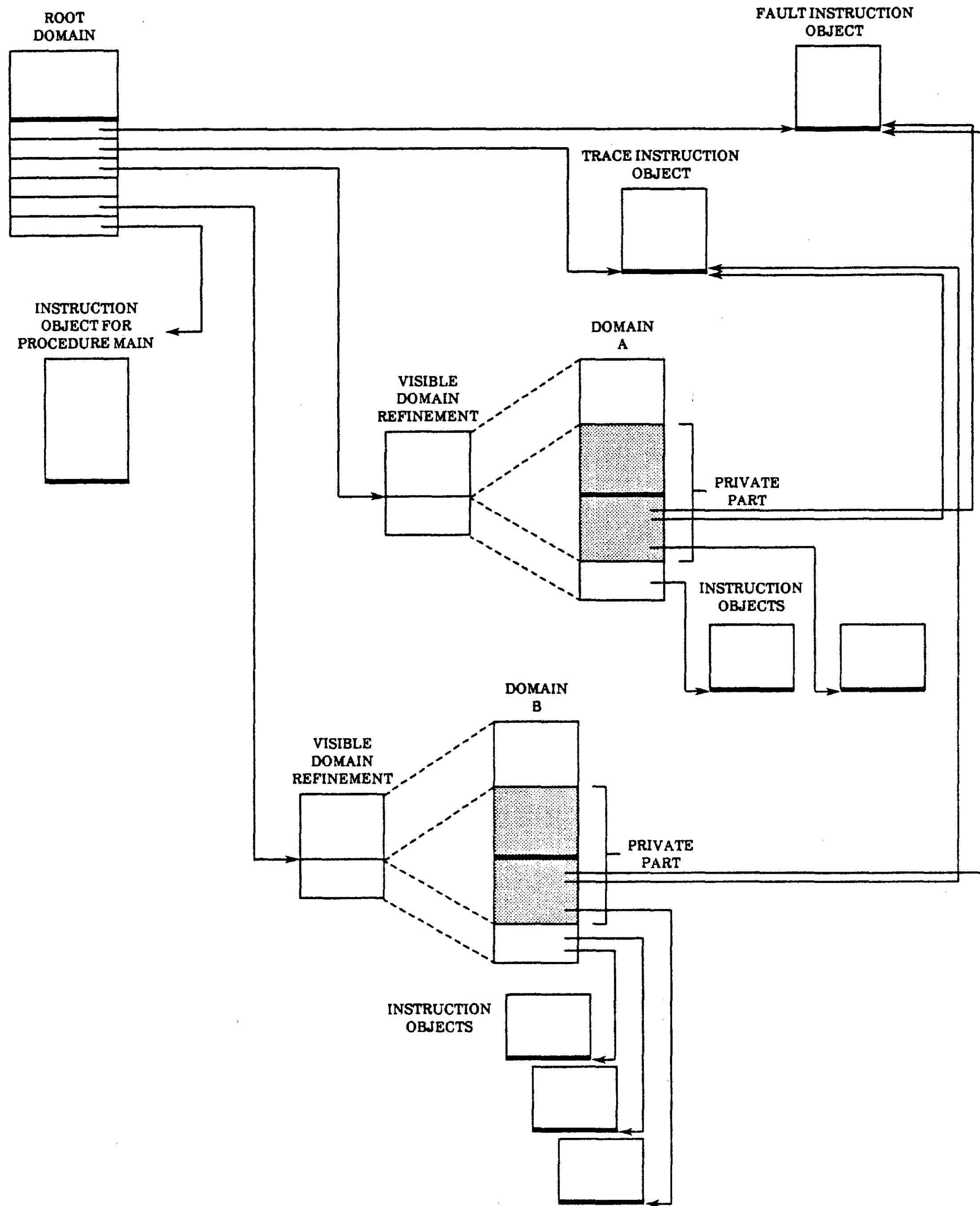


Figure 2-3. Static Program Organization Example

CONTEXT OBJECTS

A context object represents a particular call to a procedure within a domain. The context object serves two major purposes:

1. It provides storage for local variables used by the context.
2. It provides a distinct access environment for the context, giving access to those objects that the context "needs to know."

All addressing of program operands takes place from within some context, and the structure of a context is what the program "sees" when it executes. The objects that a program can access are determined by its context. Because of the importance of contexts in program organization, they are described in detail in the following sections.

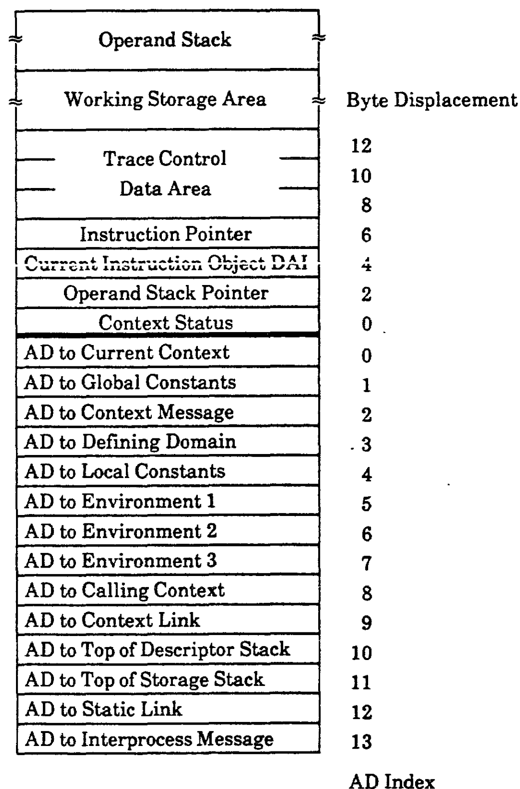


Figure 2-4. Context Object

CONTEXTS VS. PROCEDURES

It is important to distinguish between contexts and the procedures executed by contexts. There can be multiple active contexts corresponding to a single procedure. For example, three different users may simultaneously invoke a sort procedure, but for different files. In this case, three different context objects associated with three different user processes are actively executing the same code. When a procedure is recursive, it may call on itself, resulting in multiple active contexts corresponding to the same procedure and within a single process. For example, compilers frequently use recursive procedures to implement parsers.

It is enlightening to consider how contexts that correspond to the same procedure are the same and how they are different. Such contexts have access to the same instruction object and to the same domain. But the parameters referenced by each context are normally different. Also, such contexts may be executing in different processes, and attributes inherited from the process, such as standard I/O interfaces, may be different.

ACCESS ENVIRONMENT

The access environment of a context is all those objects that can be accessed from the context. The access environment is all those objects for which the context either has an AD or can get an AD by performing a series of ENTER operations. (ENTER operations are described below.) The ADs for objects within the access environment of a context may or may not have read rights or write rights; it can be very useful for a context to hold an AD for an object even if it cannot read it or write it (just how useful is described in the section "Type Managers").

One can imagine the set of ADs for all the objects in the access environment organized into a large array. An instruction executing in the context could then specify "byte 5 of the object referenced by the AD in array slot 7" as an operand. This is how an instruction specifies an operand, by an index into the access environment to specify an AD for an object, and then by an offset to a field within the data part of the object. The index into the access environment is called an access selector, because it selects an access descriptor.

The access part of an object matches this description of a "large array" of ADs; one object can contain up to 2^{14} (16,384) access descriptors. The access part of the context object itself is a major part of the access environment, and a program can reference any other object referenced by the context. An access selector is a double-byte value, and its 16 bits can select from not one, but four different access lists to specify an AD. The lower 2 bits of an access selector specify one of the four "environments," and the upper 14 bits specify an AD in the access part of the object selected by the lower 2 bits. Figure 2-5 illustrates an access selector.

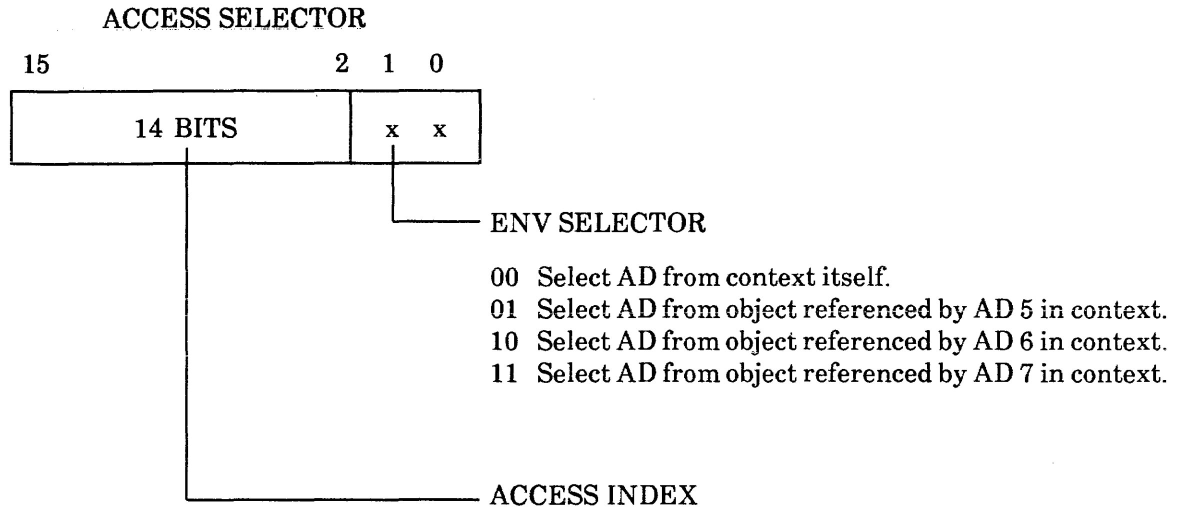


Figure 2-5. Access Selector

The network topology of ADs and accessible objects within a context cannot be directly mapped by a simple array of ADs. Consider a simple linked list of objects, in which the context directly references object A which references object B which references object C, etc. The access list in the context contains an AD for A, but not for B or C. The program must alter its own access environment before it can address data in objects B or C. This is done by executing an ENTER ENVIRONMENT operator that changes the object used for one of the access lists used in access selection. Figure 2-6 shows the example of linked objects A, B, and C, after A has been entered as environment 2.

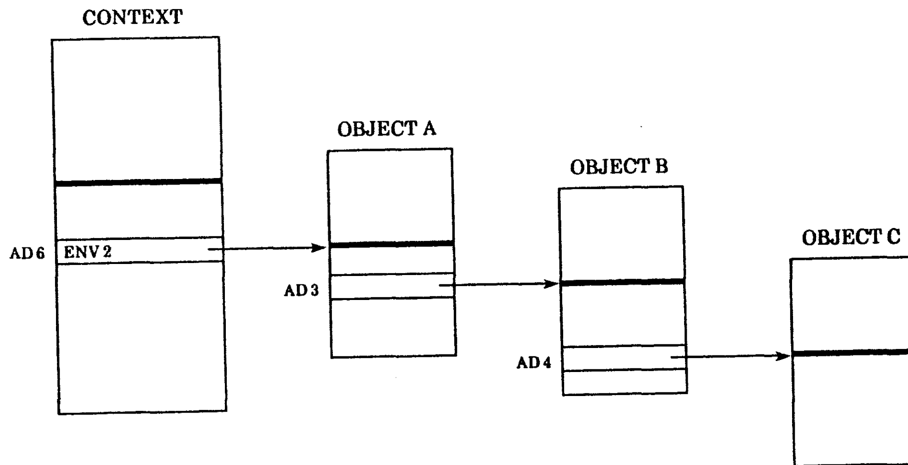


Figure 2-6. Access Environment Example

Object A is directly referenced from AD 6 of the context object. Remember that the context itself is always environment 0, which cannot be changed. An access selector value of (6,0) would be used to reference an operand in the data part of object A. Object B is referenced from AD 3 of object A, which is entered as environment 2. Because object A is entered as an environment, the data parts of any objects that A references are accessible (given proper rights). Thus an access selector value of (3,2) can be used to reference an operand in the data part of object B. Object C is not directly accessible; it is in the indirect access environment of the context but not the direct access environment. To make object C accessible, object B must be entered as an environment, e.g., as environment 3. This can be done with the operator ENTER ENVIRONMENT 3. The operand is an access selector specifying the AD to be entered, (3,2).

The three modifiable environments are a limited resource that must be managed by compilers for iAPX 432 programs. The compiler must keep track of the changing direct access environment and generate needed ENTER instructions so that all instruction operands are directly addressable when needed.

Figure 2-7 illustrates the system-defined objects in the context access environment. Many of these objects are described in the following section, "Context Description."

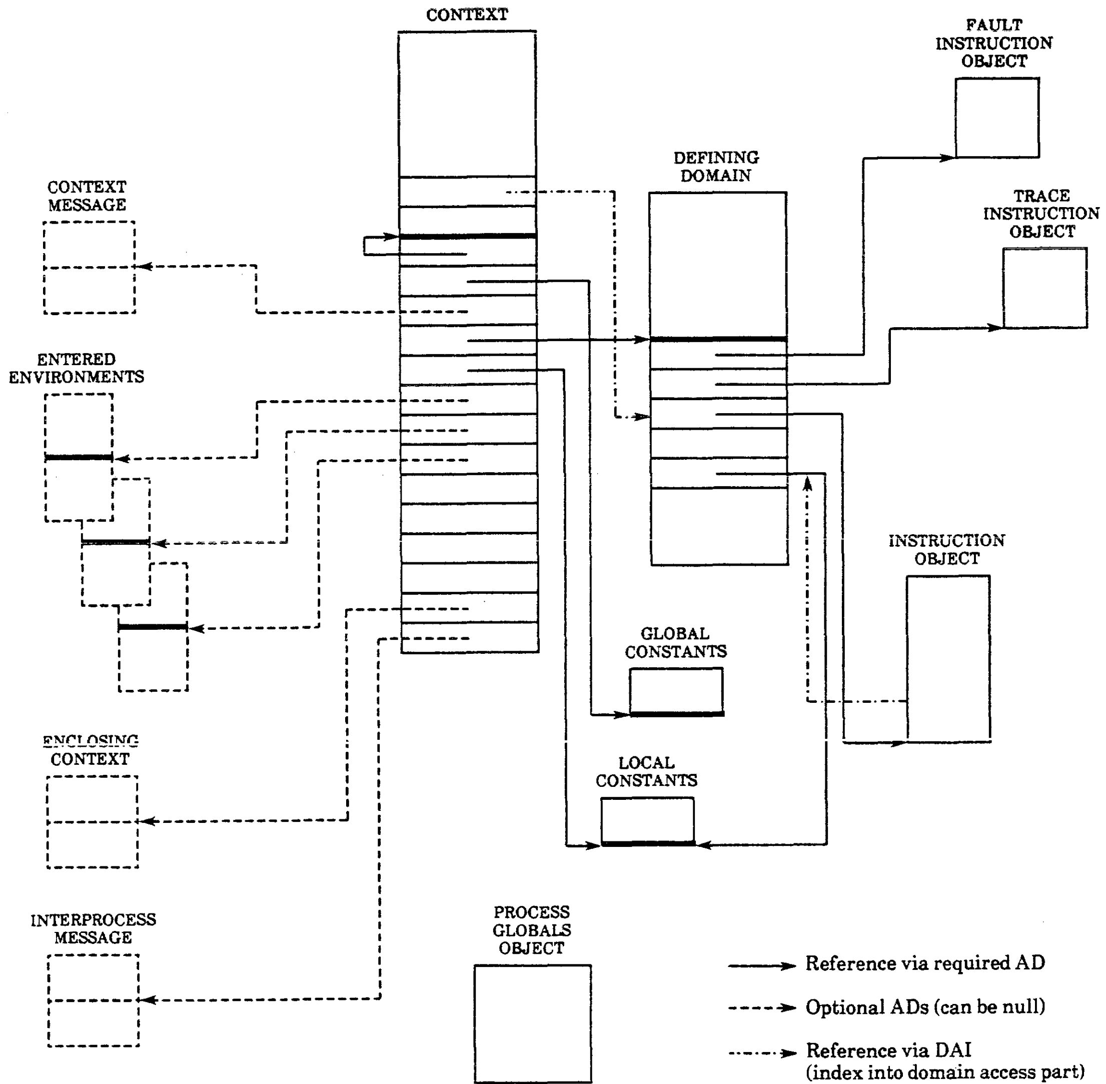


Figure 2-7. Context Access Environment

CONTEXT DESCRIPTION

This section describes the fields of a context. The context object is described in detail because the structure of a context is a large part of what a program "sees" when it executes.

Current Context (AD 0)

This AD references the context itself. This seemingly superfluous circular reference is needed for programs that reference the context data part. An access selector value of (0,0) selects this AD and the context data part.

Global Constants (AD 1)

This AD references an object containing frequently used data constants. iAPX 432 instructions cannot contain constants (literals), and all constants used by a program must be allocated in some object. The Global Constants object is a system-wide repository for frequently used constants, which then do not have to be replicated in each domain that needs them. All contexts in an iAPX 432 system should reference the same Global Constants object. Compilers for iAPX 432 systems need to know what constants are in this object, so that they can generate references to it rather than create more local constants.

Context Message (AD 2)

This AD references a refinement of the calling context (if any). The refinement is used for parameters to this context and to store results being returned to the calling context.

Defining Domain (AD 3)

This AD references the entire domain containing the procedure being executed by this context. Note that even if the procedure was called via a refinement (public part) of the domain, the AD stored here gives access to the entire domain.

Local Constants (AD 4)

This AD references an object containing data constants used by the called procedure. A compiler can create a unique local constants object for each procedure within a package, or use one local constants object for all procedures in a package. The compiler writer must consider tradeoffs in instruction length (shorter offsets may be used in some instructions if each procedure has its own constants object) versus the space overhead of additional objects (28 to 35 bytes per object).

Environment 1 (AD 5)
Environment 2 (AD 6)
Environment 3 (AD 7)

Each of these three ADs can reference an object with an access part that is entered as part of the current access environment. The access environment and the objects referenced by these ADs can be changed by executing an ENTER ENVIRONMENT operator or a COPY PROCESS GLOBALS operator. These ADs do not have delete rights and cannot be modified using the normal COPY AD or NULL AD operators. This is understandable because modifying the access environment requires changing information cached in the GDP and written in the process object; it is more than just copying an AD.

When a context is called, the defining domain is entered as environment 1 as part of the CALL operation. The ADs for environments 2 and 3 are initially null.

Calling Context (AD 8)

This AD references the calling context object. If there is no caller, i.e., if this is the initial context of a process, then this AD is null. This AD does not have read or write rights; the only part of the caller's context that can be accessed is the refinement referenced by the Context Message AD (AD 2). This AD normally has Return Rights, indicating that the caller can be returned to without faulting.

Context Link (AD 9)

This AD is a forward link to the next context in the list of preallocated contexts for the process containing this context. Preallocated contexts are discussed below.

Top of Descriptor Stack (AD 10)

This AD is used by memory management. It references the object most recently allocated from the stack, either by this context or its calling contexts. This AD does not have read or write rights, because the object it references may not be one that this context should be able to access.

Top of Storage Stack (AD 11)

This AD is used by memory management. It references the storage object most recently allocated from the process allocation stack, either by this context or its calling contexts. The difference between this AD and the Top of Descriptor Stack AD is that the latter may reference a refinement object or interconnect object; this AD references the most recently created object that was allocated storage. This AD does not have read or write rights, because the object that it references may not be one that this context should be able to access.

Static Link (AD 12)

This AD is a parameter to the CALL instruction. It is intended for use by compilers when one procedure definition is textually enclosed within another procedure. In many high-level languages, including Ada, the enclosed procedure is able to refer to all the local variables of the enclosing procedure. The context of an enclosing procedure is not necessarily the calling context, as shown in Figure 2-8.

Procedure A textually encloses Procedures B and C. Both B and C should be able to access the local variables of the call to A above them in the call chain. A calls B which calls C. For both B and C, the static link references a context for A. For B, this is also its calling context (though the static link AD has access rights and the Calling Context AD does not). For C, the static link references A even though A is not its caller.

It is relatively rare for procedures to textually enclose other procedures, and the static link parameter is often unused and null.

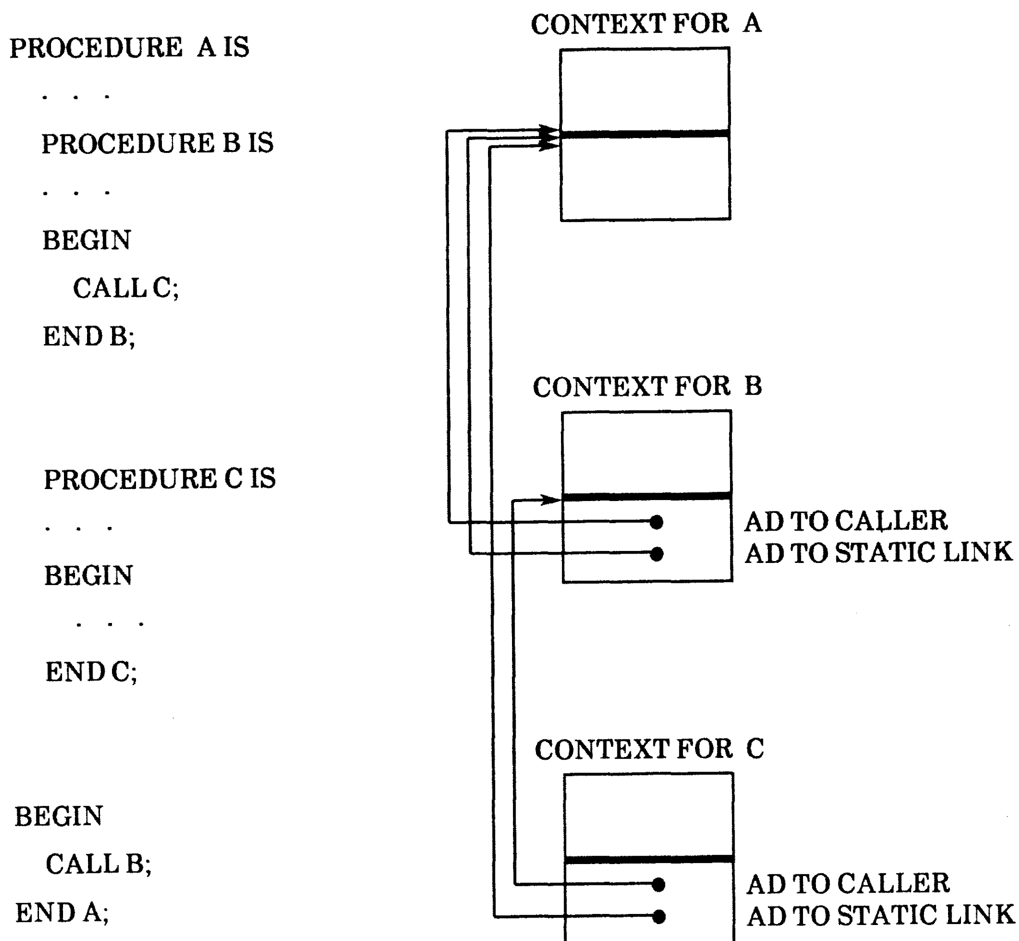


Figure 2-8. Nested Procedures Example

Interprocess Message (AD 13)

This AD references the most recent interprocess message received by the context. This AD is nulled when the context is called and until any message is received. Interprocess communication is described in Chapter 5, "Parallel Processing."

(Subsequent fields described are in the context data part.)

Context Status (bytes 0, 1)

This field contains two subfields that control precision and rounding for the GDP's floating point operators. These subfields are described in Chapter 8, "Computational Data Types."

Operand Stack Pointer (bytes 2, 3)
Operand Stack

The operand stack pointer is a byte offset into the context data part to the first free byte of the operand stack. This is an expression-evaluation stack; each context has one. This stack is used for intermediate values by computational operators. This stack is not used for procedure linkage. Using the operand stack shortens instructions and improves performance. This stack is aligned on double-byte boundaries; the operand stack pointer is always even. The operand stack grows upward in the context data part, from an initial stack pointer value specified by the instruction object header up to the end of the context data part. The bounds check automatically performed by the iAPX 432 on all object references thus provides a check on stack overflow. The stack pointer is cached by the GDP while the context is executing; thus this field is not defined during execution of the context.

Current Instruction Object DAI (bytes 4, 5)

This field is a domain access index; the upper 14 bits are an index into the defining domain's access part to the AD for the current instruction object. This value is cached by the GDP and is not defined during context execution; it can be changed during execution by an intersegment branch.

Instruction Pointer (bytes 6, 7)

This field is the bit displacement into the instruction object data part to the next instruction to be executed. The instruction pointer is cached by the GDP and is not defined during context execution.

Trace Control Data Area (bytes 8 .. 13)

This area contains control information used in tracing, and is described in Chapter 12, "Fault and Trace Reference."

Working Storage

The area (if any) between the Trace Control Area and the beginning of the operand stack can be used by the compiler to allocate local variables for the context. ADs in the context access part above the processor-recognized part can be used for local access variables.

PREALLOCATED CONTEXTS

Calling a procedure is a very frequent programming operation, and is even more frequent in programs that use the modular techniques supported by the iAPX 432. Each iAPX 432 procedure call requires a new context object, containing several specific ADs, to establish a new access environment. To make the procedure call mechanism as efficient as possible, the iAPX 432 architecture presumes that contexts are preallocated by the operating system. Each process is presumed to have a list of these preallocated contexts assigned to it when the process is created. When a procedure is called, the next free context on the process's list of contexts is used. This eliminates the overhead of creating and deleting context objects with each call and return. Also, ADs in the context that do not change between calls, such as the Global Constants AD, can be already assigned on entry, saving more time. Figure 2-9 shows a process and its preallocated contexts. Some of the contexts are active (associated with procedure calls that have not yet returned); others are not being used.

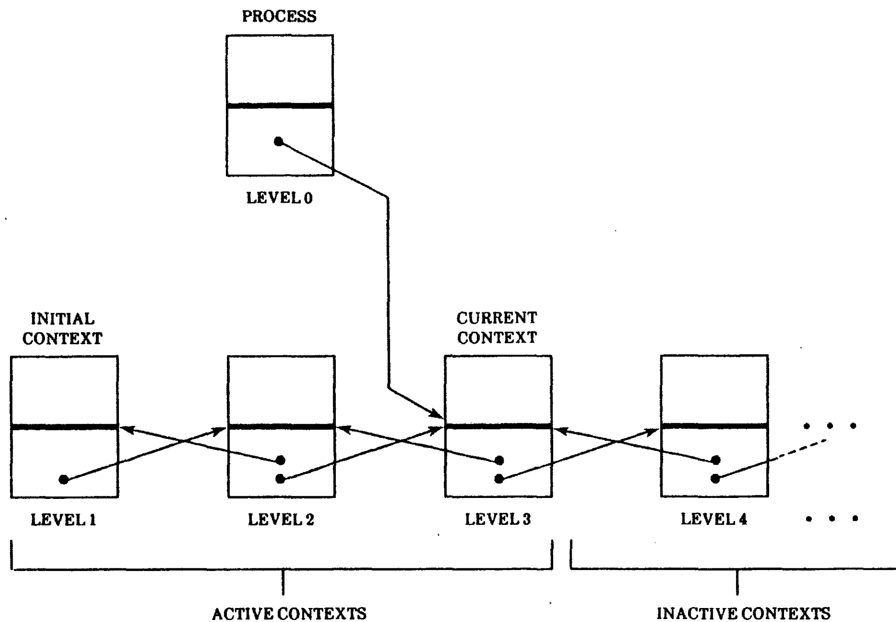


Figure 2-9. Preallocated Contexts Example

THE CALL OPERATORS

To call a procedure, the calling program must specify the domain being called, an access index into that domain to select the instruction object being called, and an AD to be passed as the static link to the new context. The CALL and CALL THROUGH DOMAIN operators differ only in how they specify the domain being called. For CALL, an AD for the new domain must be directly accessible. That is, if the new domain is referenced by the defining domain, then the defining domain must be entered as an environment. For CALL THROUGH DOMAIN, an AD for the new domain must be in the defining domain, and a domain access index is specified; CALL THROUGH DOMAIN does not require the defining domain to be entered as an environment.

Calling a procedure traverses any domain refinement (public view) used to access the procedure, and an AD for the entire domain is written into the new context. The entire domain is also entered as environment 1 by the call operation.

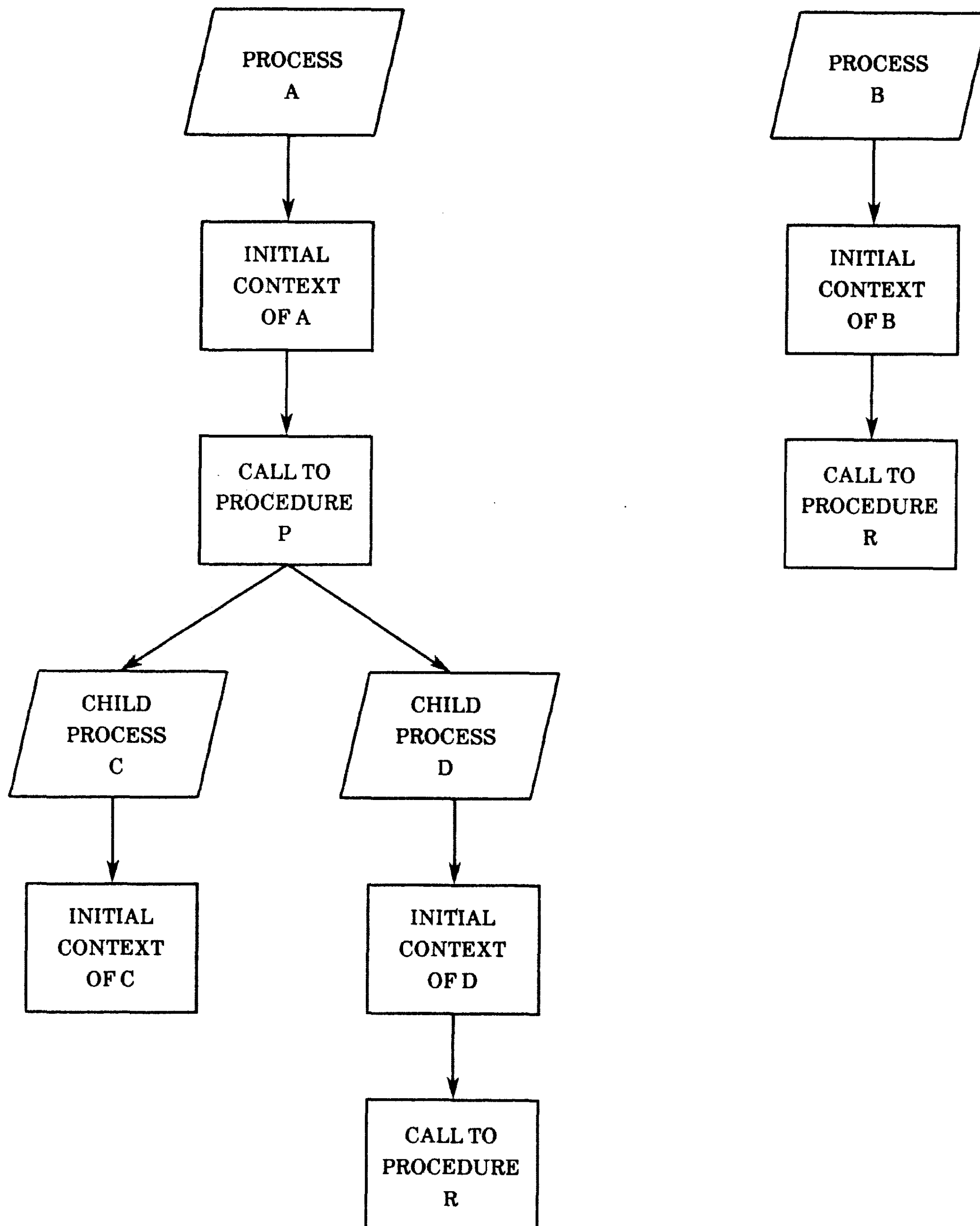
There are many other details in the initialization of a called context, described in Chapter 10, "Operator Set."

THE RETURN OPERATORS

Executing the RETURN operator automatically deallocates any stack objects created by the current context. The GDP then loads execution information from the caller (its environments, instruction pointer, stack pointer, etc.) and resumes execution within the calling context. The RETURN AND FAULT operator executes the RETURN operation and then immediately raises the Return Fault.

CONTEXT LEVEL NUMBERS

Each context in an iAPX 432 system defines the scope of objects local to the context. The scope of an object is recorded as a level number in the object's descriptor. Objects with level number 0 are global and have indefinite lifetimes. Objects with level numbers greater than zero are local to the context with the same level number. When a context is called, it has a level number that is one greater than the level of its caller. When control returns from a context, all objects local to that context (with the same level number) are deallocated. Figure 2-10 gives an example of dynamic program organization with several levels of processes and contexts. Figure 2-11 shows the corresponding object scopes and level numbers.



NOTE: Arrows indicate relationships within the program but do not correspond to ADs.

F-0294

Figure 2-10. Dynamic Program Organization Example

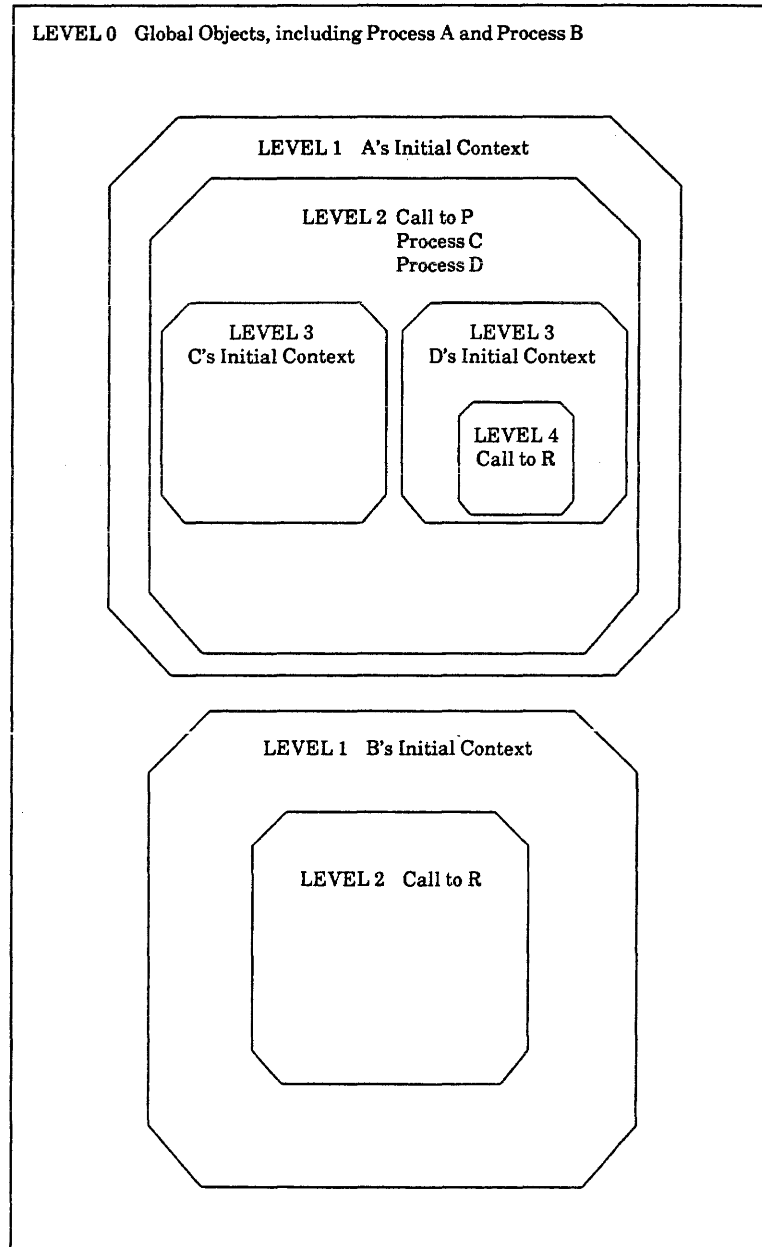


Figure 2-11. Object Scopes and Level Numbers in the Dynamic Program Organization Example

PROCESS OBJECTS

A process object represents a unit of potentially parallel activity. For example, if there are three processes in a system, then potentially all three can execute concurrently. It is natural for processes to correspond to real activities that occur concurrently. For example, if there are four terminal users simultaneously on a timesharing computer system, then each can be represented by a separate process. A process can also be associated with each physical I/O device. Thus the printer and the card reader can operate concurrently because there are separate printer_handler and card_reader_handler processes that can execute concurrently and control both devices.

Processes and multiprocessing in the iAPX 432 architecture are described in detail in Chapter 5, "Parallel Processing." Processes are mentioned in this chapter for two reasons: First, processes are part of the dynamic organization of an executing program. Second, a process contributes to the access environment of every context executing within it, via its associated process globals object.

The process globals object of a process is an object that is part of the access environment of every context executing within the process. The process globals object can be entered as an environment by using the COPY PROCESS GLOBALS operator. The process globals object can be used by an operating system; for example, the process globals object can reference a default global heap SRO to be used within the process to create global objects, or can reference standard I/O devices to be used within the process.

OBJECT MANAGERS

Consider an iAPX 432 system with a single mass storage device and a single file directory. It is desirable to hide details of the directory representation from user programs so that they do not depend on a particular representation. For example, the directory structure might change from a simple linear structure in one release to a "hashed" structure in a subsequent release, giving faster access to directory entries but requiring more main memory and more disk space. A "Disk_Manager" package can conceal the directory representation in the package body and provide an external interface with such user operations as "Create," "Open," "Read," "Write," and "Close." Note that subordinate packages used by the disk manager can conceal the representation of free blocks on the disk (bit map, linked list, or other) and of the files themselves (organization as randomly scattered blocks or as "extents" of contiguous blocks).

This type of package can be called an object manager; the module conceals the representation of one or more specific objects while providing services to external callers that use those objects.

TYPE MANAGERS

Next, consider a more difficult and more general problem in modular programming, a system with multiple file directories associated with multiple users, in which file directories can be created and deleted at run-time and can exist in complex hierarchies. Further, access to file directories must be

controlled, so that access by one user to files of another user or of the system administrator is controlled. For example, these are all attributes of the UNIX* operating system's directory structure. Finally, the solution to the problem should conceal the representation of directories just as well as the Disk_Manager package described above does in a simpler system.

A key part of the new problem is that software that uses the new "Directory_Manager" module must be able to refer to directories. For example, a caller of the "Open" operation must specify the directory containing the file to be opened. The "Create_Directory" operation must return such a reference to its caller.

Directory references held by users could be represented as index numbers. Within the directory manager, inaccessible to external callers, could be a large table of ADs for all the directories in the system. When a user created a directory, an empty slot (e.g., slot 5), would be found in this table. An AD for the new directory is then written into that slot and the slot number (5) is returned to the caller. Subsequent user calls, such as "Create file MEMO in directory 5," specify the directory by giving the slot number. The user software never has an AD for the directory and can never access the representation of the directory.

The flaw in this design is that while the directory ADs are protected, the slot numbers are not. There is nothing to prevent a user program from doing file operations in directory 8 when it should be restricted to directory 5. The slot number parameter is simply an ordinal value supplied by the caller; the caller could even systematically supply values of 0, 1, ... to access the entire file system including files of other users. The same concerns apply to any design in which the directory reference returned to the user is unprotected and corruptible.

The iAPX 432 provides only one type of "data" that is protected and incorruptible -- the access descriptor. Thus it makes sense to search for a solution that returns an AD as a directory reference. Most straightforward is to return an AD for the directory itself, <u>but without read or write rights</u>. Such a reference is protected, incorruptible, and unambiguous, but does not allow the holder to read or write the representations of directories. The body of the directory manager must map this AD without read or write rights to an AD with read and write rights for the directory. The body could do this with a look-up table of all directory ADs, all with read and write rights. This table can be scanned and the ADs compared in turn to the AD without rights supplied by the caller. The EQUAL ACCESS operator can be used, which returns true if two ADs reference the same object, even if the rights bits differ. Note that if the caller mistakenly supplies an AD for some object other than a directory, no matching entry would be found in the directory table, and an exception could be raised.

Such a module, which provides all operations on a particular class of object, but allows other modules to hold references without rights to such objects, is called a <u>type manager</u>.

*UNIX is a trademark of Bell Laboratories.

TYPE MANAGER IMPLEMENTATION

A type manager can actually be implemented without the look-up table mechanism described above; the GDP provides an AMPLIFY RIGHTS operator that takes an AD without rights and "turns on" selected rights (such as read and write rights). Amplifying rights must be a privileged operation, or it would be meaningless to restrict read or write rights at all. The AMPLIFY RIGHTS operator requires an AD (with amplify rights) for a Type Control Object (TCO) that specifies the type of object for which rights can be amplified. The TCO also specifies which rights can be amplified. In the directory manager example, the body of the module can access a TCO for directories, giving it the privilege of amplifying rights on ADs for directory objects. This TCO is not available to external users of the module.

The implementation of type managers uses four features of the iAPX 432 architecture:

1. GDP support for protected program modules (domains), where external users can only access a refinement of the domain, but code in the body of the module has access to the entire domain.

2. GDP support for software-defined protected types.

3. GDP implementation of creating typed objects as a privileged operation.

4. GDP implementation of amplifying rights to typed objects as a privileged operation.

SOFTWARE-DEFINED PROTECTED TYPES

In the hypothetical "look-up table" implementation of the type manager, described above, the table itself provided a secure "typing" of directory objects. An object was a directory if and only if it could be found in that table. Only the type manager module had access to the table, thus only the type manager could create an object as a directory object by creating it and entering an AD for it into the table. The actual implementation of type managers uses the AMPLIFY RIGHTS operation to map ADs for the managed objects without rights to ADs for the managed objects with rights. There is no look-up table, but a secure typing of directory objects is still required. Otherwise, an external caller could spoof the directory manager by creating an object and passing it to the manager in the guise of a directory object. At best this might crash the system; at worst it might introduce errors into the file system or give the caller access to protected files.

Remember that all information in an iAPX 432 system is represented as objects; thus the iAPX 432 represents types as objects as well. A software-defined protected object type is represented by a type definition object (TDO). The TDO has no processor-recognized fields. The TDO data part might contain a printable name for the type (e.g., "directory") and the TDO access part might reference the type manager domain for the type.

Each object of the protected type is represented as a dynamic type object (DTO). A DTO has no processor-recognized fields but is formatted by the type manager. For example, a directory object might contain file names, disk addresses, and file protection information. The object descriptor for a DTO contains a copy of an AD for the TDO that defines the type. The GDP operator RETRIEVE TYPE DEFINITION takes an AD for a DTO and returns an AD for the TDO that defines its type.

Figure 2-12 illustrates the relationships between the objects used to implement a type manager: domain, TDO, TCO, and DTOs.

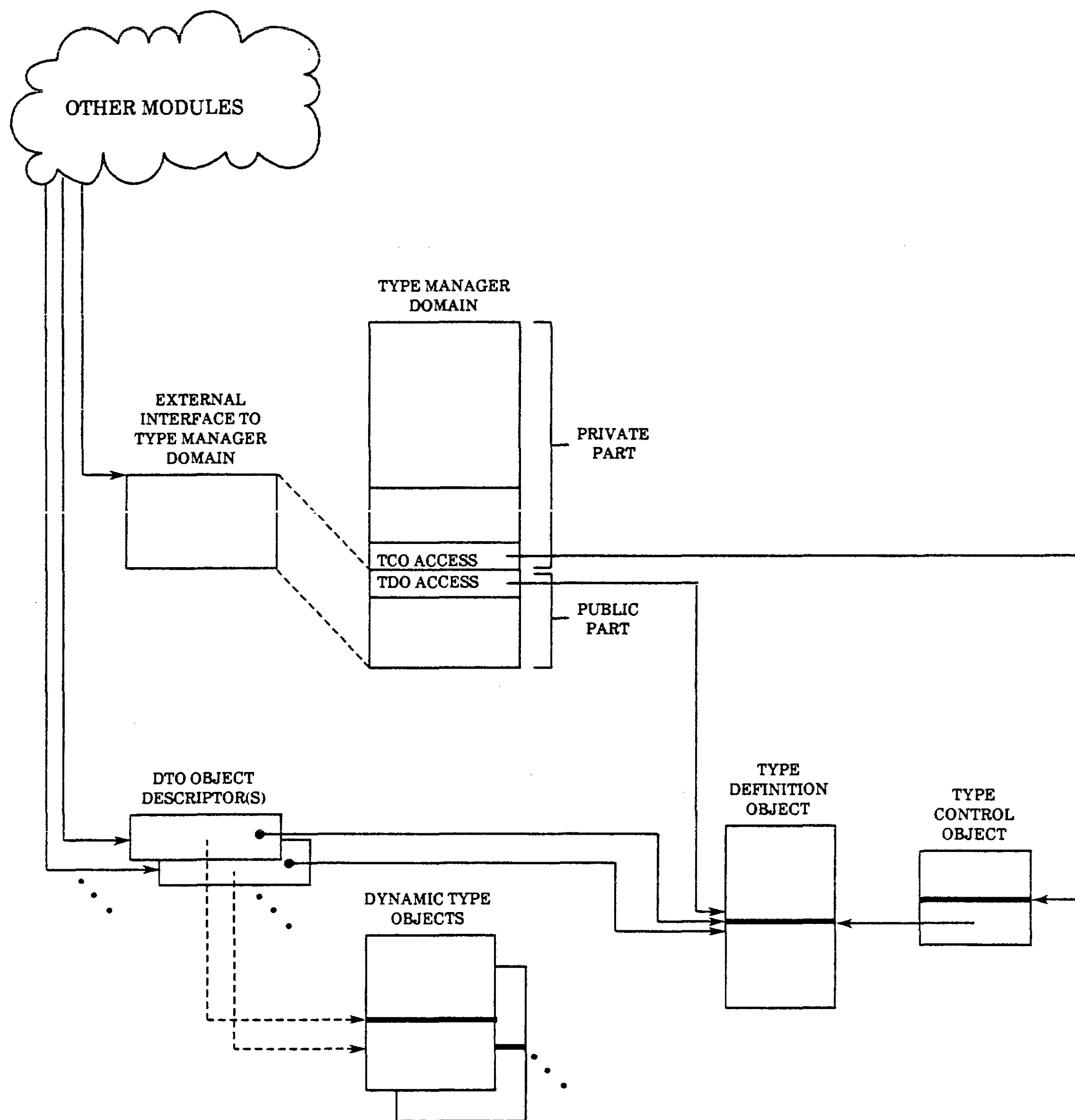


Figure 2-12. Type Manager Objects

It is important to clearly understand the difference between a Type Definition Object and a Dynamic Type Object. Suppose that there are two software-defined types in a particular system, "directory" and "user_job", and that there are 78 directories and 10 user jobs at a particular instant. The system will contain two TDOs corresponding to the two software-defined types. The system will contain 88 DTOs. 78 of these DTOs will represent particular directories. Executing the RETRIEVE TYPE DEFINITION operator for any of these 78 DTOs will return an AD for the "directory" TDO. Ten of the DTOs will represent particular user jobs. Executing the RETRIEVE TYPE DEFINITION operator for any of these ten DTOs will return an AD for the "user_job" TDO.

CREATING TYPED OBJECTS

Creating objects of a protected type must be restricted to the type manager for two reasons. First, when a typed object is created, initial values for its contents must be written into it, a privileged operation that accesses the object's representation. Second, when an object is created by the GDP, the returned AD has all rights. Read and write rights must be removed from the AD using the RESTRICT RIGHTS operator, before returning the AD to a caller from outside the type manager.

The CREATE TYPED OBJECT operator is used to create an object of any type other than "generic." This operator requires an AD with create rights for a Type Control Object that specifies the type of the new object. Because only the body of a type manager has access to a TCO for the managed type, only the type manager can create objects of the type.

TYPE MANAGER SCHEMA

This section gives a schema or "template" that can be used for defining type managers. This same schema can be used to manage both system typed objects (e.g., processes) and software typed objects.

1. For a software-defined type, the package initialization code should call an operating system package that creates a new unique software-defined type. The call should return ADs for a TCO and a TDO. The TCO AD should be stored in the private part of the type manager domain and nowhere else.

2. (If the managed type is XXX), the package should provide a Create_XXX operation. Create_XXX should use the CREATE TYPED OBJECT operator to create an object of the type, then initialize the object, then use the RESTRICT RIGHTS operator to remove read and write rights, and then return the AD without rights to the caller.

3. Each operation provided on objects of the type will take one or more ADs for objects of the type as parameters. The body of each operation should use the AMPLIFY RIGHTS operator to get read and write rights for these objects within the body of the type manager. Remaining code should implement the operation.

CHAPTER 3
OBJECT ADDRESSING

This chapter describes the iAPX 432 architecture's object addressing mechanism, which provides these services:

- conversion of access selectors and offsets to physical addresses for operands
- bounds checking and checking of read/write rights for all memory references

The GDP supports several different addressing modes -- various ways that an access selector and offset can be specified by an instruction. The GDP addressing modes are described in Chapters 7, "Instruction Interface," and 11, "Instruction Encoding."

This chapter describes these iAPX 432 types:

- access selectors
- access descriptors
- object descriptors
- object tables

Access selectors are also described in Chapter 2, "Program Organization." More information about access descriptors, object descriptors, and object tables is contained in Chapter 4, "Memory Management."

PHYSICAL ADDRESS SPACES

An iAPX 432 system has two physical addressing spaces, a storage address space and an interconnect address space. The interconnect address space can be used for hardware configuration information, interprocessor communication registers, and error registers. All other information in the main memory of an iAPX 432 system is contained in the storage address space. The section "Interconnect Addressing" describes the interconnect address space. All other sections of this chapter describe object addressing in the storage address space.

Though an iAPX 432 system contains multiple processors, all processors share a single common storage address space. Even processor state information is stored in the storage address space, in processor objects.

The iAPX 432 supports a storage address space with up to 2^{24} (16,777,216) bytes of memory.

TWO-PART MEMORY REFERENCES

All iAPX 432 memory locations are accessed through some object. Even free storage blocks (storage currently not contained in an object) are referenced by descriptors in a system object that represents a free storage pool.

iAPX 432 programs never access memory using physical addresses. Instead, a program accessing memory specifies the object being accessed. Because each iAPX 432 object can contain multiple data fields, a memory reference also specifies the offset from the base of the object to the field referenced.

> Each iAPX 432 memory reference has two parts:
>
> 1. An access selector that specifies an object.
>
> 2. An offset into the object's data part to the referenced operand.
>
> The combination of an access selector and an offset is called a logical address.

TWO-LEVEL ADDRESS MAPPING

The iAPX 432 architecture maps an access selector to a physical base address for an object using two levels of descriptors (see Figure 3-1). The two-level address mapping supports two different functions: protection and dynamic storage management.

ADDRESS MAPPING FOR OBJECT PROTECTION

A fundamental part of the iAPX 432 architecture is a uniform protection mechanism for objects. Each call to a subprogram is represented by a context object that has references for only those objects that it has a "need to know." The object references are represented by access descriptors (ADs). The ADs may allow only restricted access to the objects they reference. For example, an AD may only allow a context to read an object but not to write it. Thus an AD has two parts: a unique system-wide identifier for the referenced object and rights information that indicates the operations allowed using the AD. An AD can also have the value null, referencing no object.

An AD corresponds to a high-level programming language's pointer or access values; the term "access descriptor" is derived from the Ada language's access types. Like the pointer values in high-level languages, ADs can be copied at run-time and more than one context may have ADs for the same object.

There is no way to read or write an object without an AD, not even if the object's identifier or physical address is known.

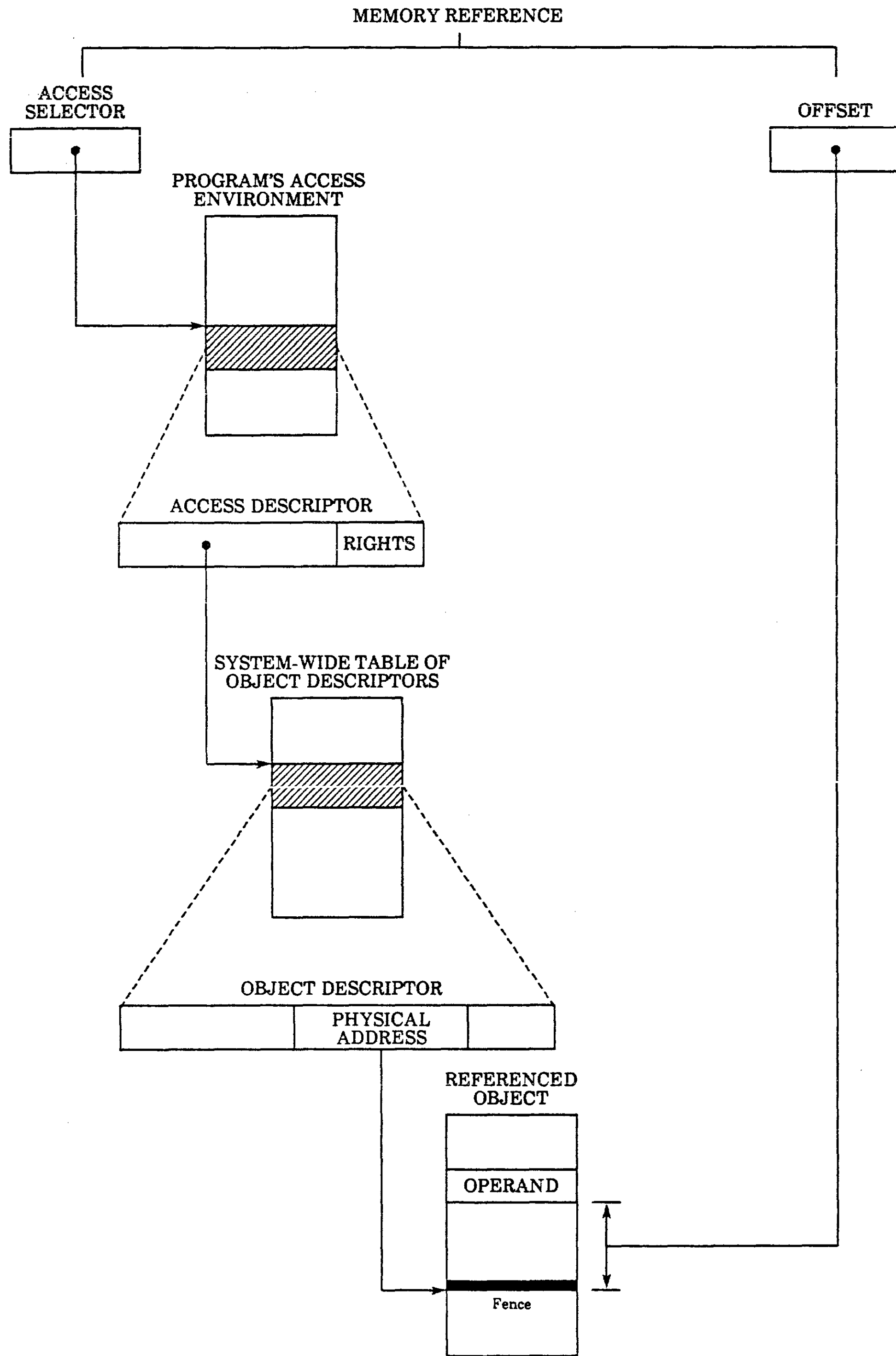


Figure 3-1. Two-Level Address Mapping

> To reference an object, a context must have an access descriptor for the object.

Because an AD gives its holder rights to reference an object, protection of objects requires protection of access descriptors. Even when a program has write rights to an object, access descriptors in the object cannot be manipulated as arbitrary bit patterns (e.g., added, subtracted, shifted, assigned). Access descriptors can only be overwritten by other access descriptor values. New access descriptors are created only by the GDP, never by software, and only when creating new objects.

Any object in the storage address space can contain both ADs and data. The ADs in an object represent relations to other objects. For example, the context object for a subprogram call contains an AD for its caller. The context object also contains data, such as local variables used by the subprogram call.

OBJECT FORMAT

Though objects can contain both data and ADs, the two kinds of information are physically segregated within an object, into a data part and an access part. The data part can contain anything but ADs; the access part can contain only ADs. Each part is optional. An object can have a data part and no access part, or an access part and no data part. Each part is limited to 2^{16} (65,536) bytes. Therefore, the total size of an object is limited to 2^{17} (131,072) bytes. Because an AD requires four bytes, the access part of an object is limited to 2^{14} (16,384) ADs. Figure 3-2 shows the object format.

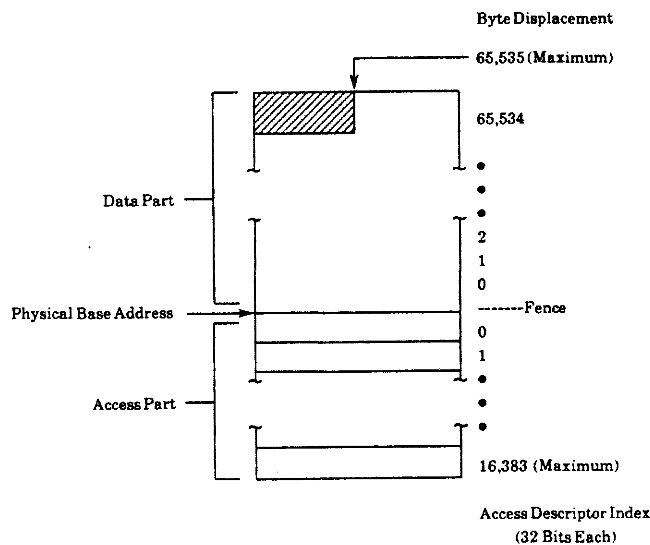


Figure 3-2. Object Format

Objects are laid out in physical memory so that the physical base address of the object is the address of the first byte of the data part. The data part occupies storage locations above the base address and the access part occupies storage locations below the base address. The base address acts as a "fence" between the two parts of an object. Note that ADs with higher index values occupy lower storage locations.

ACCESS DESCRIPTOR FORMAT

Figure 3-3 shows the format of an access descriptor.

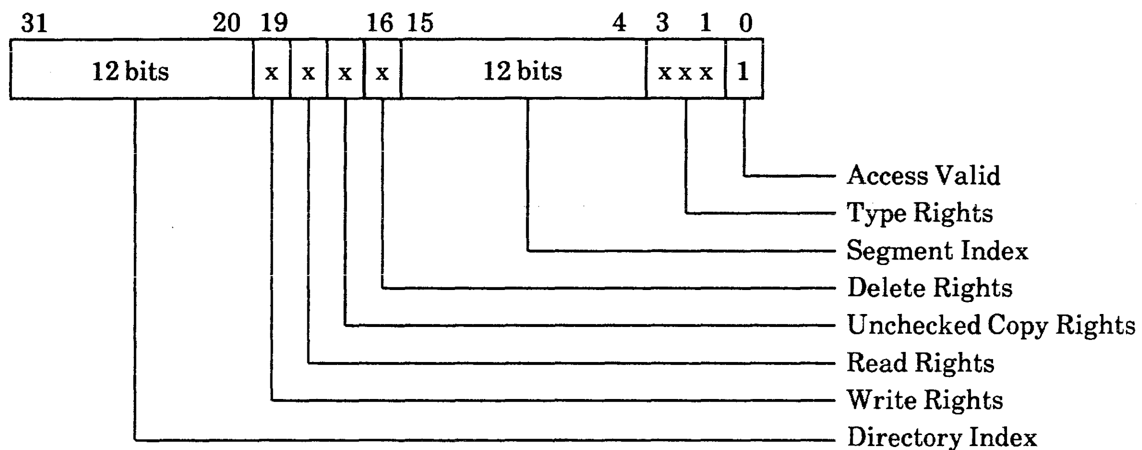


Figure 3-3. Access Descriptor Format

The access valid bit is 1 if the AD references an object and 0 if the AD is null.

The directory index and segment index fields together constitute a 24-bit object index, a unique, system-wide, unchanging identifier for the referenced object.

The various rights bits allow or restrict operations using the AD. Access rights allow or restrict operations on the AD itself. The access rights on an AD are the read rights, write rights, and three type rights. Read rights are required to read from an object. Write rights are required to write into an object. The three type rights bits are used to allow or restrict operations specific to the particular type of object referenced. For example, for port objects, type right 1 (in bit position 1) is interpreted as send rights, required to send a message to the port. Type right 2 (in bit position 2) is interpreted as receive rights, required to receive a message from the port. Of course, for a different type of object, the type rights can have a different meaning or no meaning at all.

There are two AD rights, delete rights and unchecked copy rights. Delete rights are required on an AD for it to be overwritten by another AD. If an AD does not have delete rights, then it can only be deleted by deleting the entire object containing it. Unchecked copy rights allow an AD to be copied without a level check. Level checking and the use of unchecked copy rights are described in Chapter 4, "Memory Management."

Operations on ADs, and how to use them to implement type manager modules, are described in Chapter 2, "Program Organization."

ACCESS SELECTORS

Instructions that reference objects cannot just include access descriptors in the instruction stream itself. For objects created at run-time or received as parameters or messages, the AD values are not known when the instruction object is created. Thus instructions must be able to specify an index, called an access selector, that selects the AD "slot" containing an AD for the referenced object. To preserve the strict separation between ADs and other information, instructions cannot include ADs and always specify objects using access selectors, even when the AD value is known at compile-time.

> An access selector specifies an object by specifying an AD slot in the current context that contains an access descriptor for the object.

An access selector can be embedded directly in the instruction stream or can be specified indirectly and taken from the data part of some object. Note that indirect access selectors can be assigned and modified under program control. For example, an indirect access selector can be modified in a loop to sequence through multiple objects.

ACCESS SELECTOR FORMAT

An access selector value is 16 bits and can select one of up to 2^{16} (65,536) access descriptors. Remember that the access part of any one object is limited to 2^{14} (16,384) ADs. An access selector has two parts, a 2-bit environment selector and a 14-bit access index. The environment selector selects the access part of one of four objects, called an environment when so selected. The access index selects an AD within the environment. The access part of the current context object is always one environment; ADs in the context reference the other three environments. The GDP provides special ENTER operators to make an accessible object one of the three alterable environments.

ADDRESS MAPPING FOR DYNAMIC STORAGE MANAGEMENT

The iAPX 432 architecture is designed to support dynamic storage management, in which objects can be relocated in physical memory. Objects may be relocated in a virtual memory system because they are swapped out to disk and later reloaded at a different physical address. In a real-memory system, objects can be moved by a compaction process that relocates objects to reduce memory fragmentation.

Because the physical addresses of iAPX 432 objects can change, access descriptors cannot contain the physical addresses of the objects that they reference. For any object, there could be many access descriptors and no way (except an exhaustive search of memory) to locate them all to update addresses in them.

Instead of specifying the physical address of an object, an AD references another descriptor, the object descriptor, that does. There is exactly one object descriptor per object in an iAPX 432 system. The object descriptor for an object is itself contained in an object, an object table object.

The position of an object descriptor (OD) in the structure of object tables in an iAPX 432 system is fixed for the life of the corresponding object. Even though the location of objects in physical memory can change, the positions of the associated ODs in the structure of object tables is fixed. An index into the structure of object tables to an OD, called an object index, has these desirable attributes:

- The object index does not change during the life of an object.
- The object index provides a way to find the OD for an object, and then to find the physical address of the object from the OD.
- The object index is a system-wide identifier for an object, not relative to any process or context. This means that pointers (ADs) to objects can be transferred between processes and between contexts.

> An access descriptor contains an object index for the referenced object, which selects the object descriptor for the object from the system-wide structure of object tables.

Each object descriptor is 16 bytes. An OD specifies the object's physical base address and the lengths of the data part and access part. An OD also contains type information and storage management information.

TWO-LEVEL OBJECT TABLE STRUCTURE

Because each OD is 16 bytes, the data part of an object table object can hold up to 2^{12} (4,096) ODs. The iAPX 432 architecture uses a two-level object table structure to allow up to 2^{24} (16,777,216) objects in a single iAPX 432 system. One special object table, the object table directory (OTD), contains only object descriptors for all object tables in the system. The object index that specifies an OD has two parts, directory index and segment index. The 12-bit directory index selects an object table from the OTD. The 12-bit segment index selects an OD from the selected object table. The two-level object table structure has these advantages:

- The structure increases the number of objects allowed in a system, providing 2^{41} bytes of virtual address space.
- The structure allows object tables to be dedicated to particular processes or storage pools. For example, all objects allocated from a particular storage pool are referenced by one object table and can be reclaimed together.
- The structure reduces contention between processes for exclusive access to object tables when creating, reclaiming, or relocating objects.

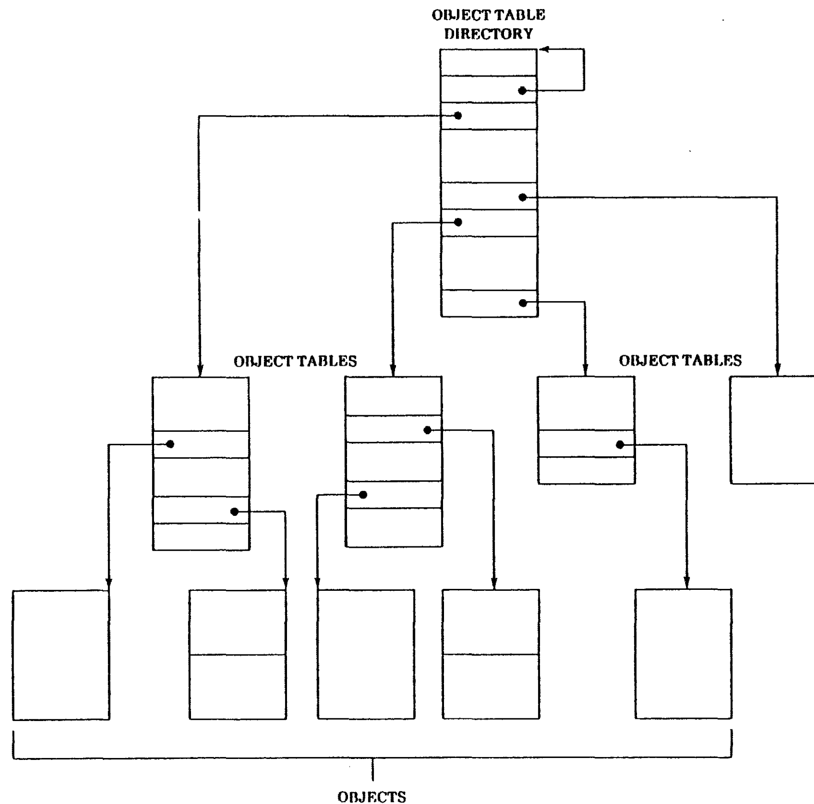


Figure 3-4. Two-Level Object Table Structure

OVERVIEW OF OBJECT ADDRESSING

This section gives a narrative overview of the iAPX 432's object addressing mechanism. Figure 3-5 illustrates the mechanism and is keyed to the narrative.

1. An operand reference in an instruction specifies a 16-bit access selector and a 16-bit offset.

2. The access selector specifies one of four objects as environments and selects an access descriptor from the access part of the chosen environment. This mapping supports the iAPX 432 object protection mechanism.

3. The access descriptor specifies access rights that determine what operations can be performed on the object with that AD. There can be many ADs for an object, each with different rights. This facility supports the iAPX 432 object protection mechanism.

4. The access descriptor also specifies a unique object index for the object. The object index selects one of up to 2^{12} object tables from the single object table directory, and also selects one of up to 2^{12} object descriptors from the selected object table. There is only one OD per object.

5. The OD specifies the physical base address of the object. The operand address is computed by adding the offset specified by the operand reference to the object's base address. Note that an operand is always in the data part of an object.

ADDRESS SPACE SUMMARY

The logical address space of an executing context consists of the directly accessible objects in its access environment (up to 2^{16} objects).

The virtual address space of an iAPX 432 system consists of all objects defined in the system (up to 2^{24} objects, up to 2^{41} bytes).

The physical address space of the iAPX 432 consists of the linear storage address space (up to 2^{24} bytes) and the linear interconnect address space (up to 2^{24} bytes).

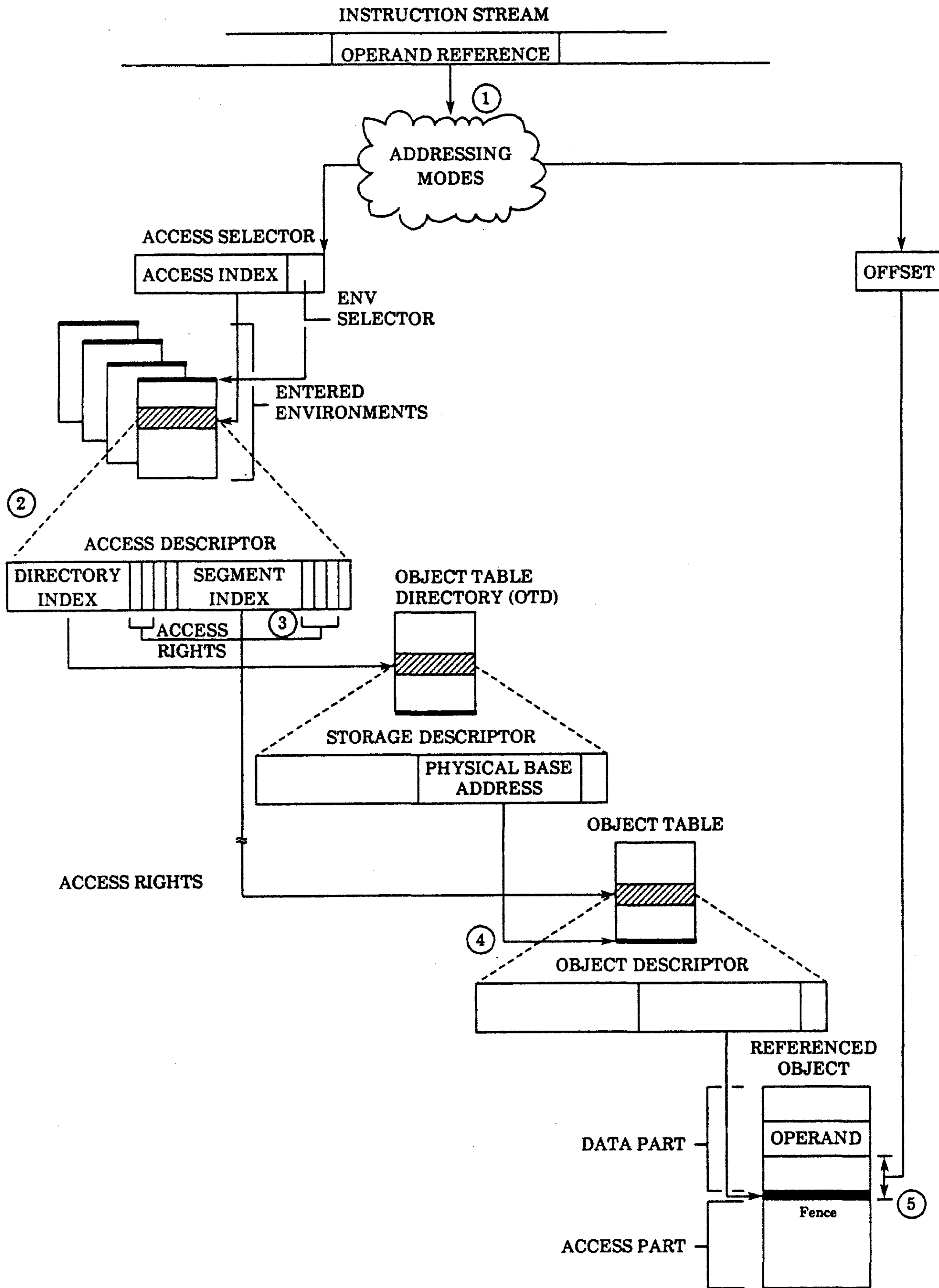


Figure 3-5. Object Addressing

REFINEMENT ADDRESSING

The iAPX 432 architecture supports the definition of refinement objects, objects that are actually part of another object called the base object. Like any other object, a refinement object can have an access part and a data part. The access part is contained in the access part of the base object and the data part is in the data part of the base object.

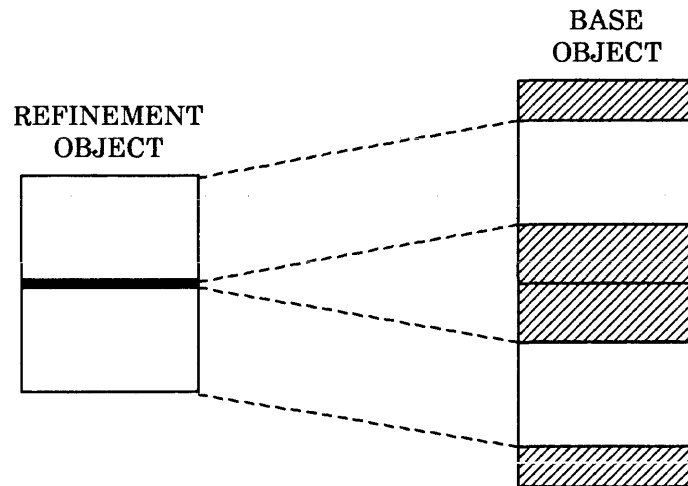


Figure 3-6. Refinement Object

A refinement object is described by a particular type of object descriptor, called a refinement descriptor. Instead of containing a physical base address, the refinement descriptor contains the object index of the object being refined. The refinement descriptor also contains offsets within the selected object to the portions accessible using the refinement, and contains the lengths of the refinement access part and data part.

INTERCONNECT ADDRESSING

The separate interconnect address space can be used for hardware configuration information, interprocessor communication registers, and hardware error registers. Use of the interconnect address space in processor communication and configuration is described in Chapter 6, "Processor Management."

All interconnect locations are accessed via interconnect objects. These objects are described by a particular type of object descriptor, called interconnect descriptors.

Interconnect objects have no access part, and access descriptors cannot be stored in the interconnect address space.

Interconnect descriptors do not specify processor type or system type for the corresponding interconnect objects. All processors can reference interconnect objects (equivalent to the processor type all), and interconnect objects can be considered similar to objects with the system type generic because interconnect objects have no processor-recognized meaning.

The interconnect address space spans up to 2^{24} (16,777,216) bytes. Interconnect objects must be aligned on double-byte boundaries (even addresses) and must contain an even number of bytes. All references to interconnect objects must use even offsets to also be aligned on double-byte boundaries.

The GDP provides operators to move a short ordinal value to or from a double-byte in an interconnect object, MOVE TO INTERCONNECT and MOVE FROM INTERCONNECT. These operators fault if a noninterconnect object is specified where an interconnect object is expected. No other GDP operators can be used to read or write operands in interconnect objects; attempting to do so causes a fault.

CHAPTER 4
MEMORY MANAGEMENT

The iAPX 432 architecture supports important memory management capabilities needed by system designers and implementers. Memory management in iAPX 432 systems is a hardware/software partnership, and the facilities described in this chapter are provided by, not just the GDP, but by the GDP architecture in cooperation with an operating system, such as iMAX 432. iAPX 432 memory management:

1. dynamically allocates new objects with single instructions.

2. completely supports the scope rules of Ada and other high-level languages.

3. automatically deallocates objects that are no longer needed.

4. supports virtual memory.

5. transparently expands free storage pools and object tables as needed by executing programs.

This chapter covers these topics:

- object scope
- system objects used for memory management
- object creation
- object lifetime strategies, which determine how objects are deallocated
- fragmentation and compaction
- virtual memory
- frozen memory
- multiple processors and memory management

OBJECT SCOPE

Each object in an iAPX 432 system has a scope, which is either global or local to some context. A global object exists indefinitely, and is only deallocated when no ADs exist for the object. (Because objects can only be accessed via ADs, and because ADs for existing objects can only be copied, not created, then once all ADs for an object are deleted, the object can never again be accessed and can therefore be reclaimed.) Objects local to a context (to a subprogram call) can only be accessed from within that context or subordinate contexts. The iAPX 432 architecture guarantees that access descriptors for local objects cannot be exported out of their scope. Thus, when a context returns, all objects local to that context can be deallocated.

Object scope is indicated by a level number in the object's descriptor. Objects with level number zero are global and have indefinite lifetimes. Objects with level numbers greater than zero are local to the context with the same level number. When a context is called, it has a level number that is one greater than the level of its caller. The section "Context Level Numbers" in Chapter 2, "Program Organization," gives an example of program organization with several levels of processes and contexts, showing object scopes and level numbers.

Whenever an access descriptor is copied, a level check is normally performed, to verify that the destination object has a level number greater than or equal to the level number of the object referenced by the AD. This ensures that the scope of the destination object is the same scope as, or is contained within, the scope of the object being referenced, and prevents ADs for objects from being exported out of their scope. For example, a context cannot return an AD for an object local to it to its caller. If a level check fails, the AD is not copied and the Level Fault is raised.

Because ADs for global objects will always pass a level check, these ADs are created with unchecked copy rights, which enables them to be copied without a level check, saving execution time.

OBJECTS FOR MEMORY MANAGEMENT

This section describes how four types of system objects are used in memory management:

- storage resource objects, reference all the other objects needed for object allocation; the other objects provide object descriptor space, physical memory space, and operating system authorization for allocation.
- object tables, contain object descriptors and free space for allocating new ODs.
- physical storage objects, reference free blocks of physical memory.
- storage claim objects, enforce operating system limits on memory allocation by a particular process or group of processes.

Figure 4-1 shows these objects.

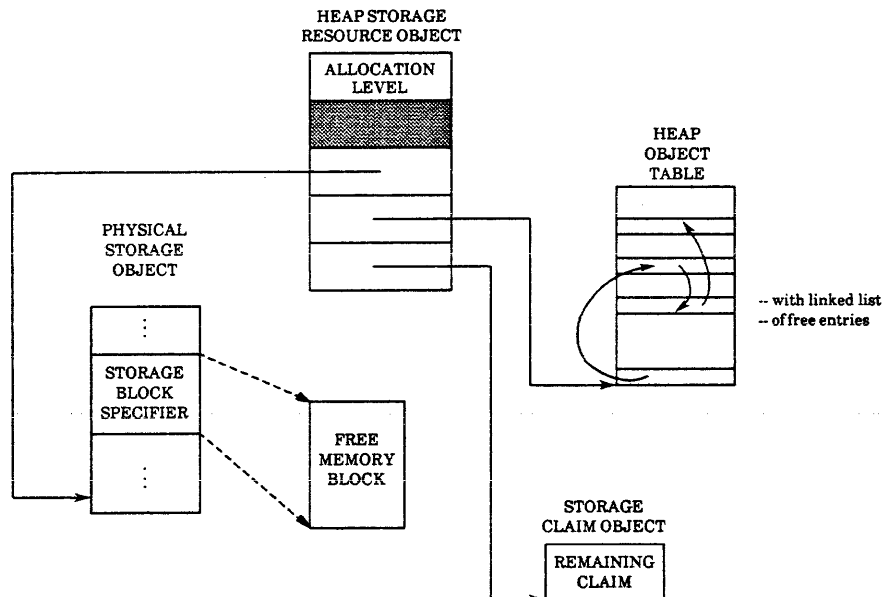


Figure 4-1. Objects for Memory Management

STORAGE RESOURCE OBJECTS

Creating a new iAPX 432 object requires 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 iAPX 432.
- a block of contiguous physical storage for the new object -- this is allocation of the physical address space of the iAPX 432.

Note that creating an object refinement 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. There are two types of SROs, heap SROs and stack SROs. Objects created from heap SROs may be deallocated by garbage collection (described below), whenever no more ADs for the objects exist. Because the order that heap objects are deallocated is unrelated to the order that they are created, allocated and free memory blocks and allocated and free object table entries may be interleaved in a heap SRO. In a stack SRO, the last objects allocated are the first objects deallocated, and deallocation is done automatically by the GDP. Because objects are allocated and deallocated in this nested fashion, free memory and free entries in the stack object table, are maintained as single contiguous blocks. Each process contains a single stack SRO; there is no support for stack SROs apart from some process. While conceptually a distinct object, the stack SRO is contained in the process object, and designated by a special access selector value in the instructions that create objects. Only heap SROs can be referenced by ADs.

OBJECT TABLES

Each SRO references a distinct object table, used to allocate object descriptors for objects created from the SRO. SROs do not share object tables. Heap object tables, referenced by heap SROs, have a slightly different structure than stack object tables, referenced by stack SROs, as described in Chapter 9, "Object Set."

PHYSICAL STORAGE OBJECTS

A physical storage object (PSO) contains storage block descriptors that delimit free blocks of physical memory. Physical memory is allocated from a heap PSO using a rotating first-fit algorithm. A stack PSO should only reference a single free block, used for all allocations and deallocations for a stack SRO. A stack SRO references a distinct stack PSO, and stack SROs cannot share a PSO. Many heap SROs can share a single heap PSO. The structure of stack and heap PSOs is identical, with stack PSOs simply referencing a single free block while heap PSOs can reference multiple free blocks.

STORAGE CLAIM OBJECTS

Heap SROs can reference a storage claim object (SCO) that limits the number of bytes of physical memory that can be allocated from the heap SROs that reference it. Multiple heap SROs can reference the same SRO. If a heap SRO's AD to Storage Claim Object is null, then there is no limit (except the available physical memory) on the amount of storage allocated via that SRO. Stack SROs cannot reference an SCO; however, the size of the single free memory block used by a stack SRO is a limit on the number of bytes allocated via the stack SRO.

OBJECT CREATION

The iAPX 432 operator set includes operators to create new objects and return ADs for them, CREATE OBJECT, CREATE TYPED OBJECT, CREATE REFINEMENT, and CREATE TYPED REFINEMENT.

CREATE OBJECT creates a generic object with processor type all. CREATE REFINEMENT creates a generic refinement with processor type all. CREATE TYPED OBJECT is a privileged operation and requires an access with create rights to a Type Control Object (TCO); the new object has the object type specified by the TCO. CREATE TYPED REFINEMENT is a privileged operation and requires an access with refine rights to a Type Control Object (TCO); the new refinement has the object type specified by the TCO. Both CREATE TYPED OBJECT and CREATE TYPED REFINEMENT are normally used only by type managers for whatever type of object is being created.

All of the CREATE operators can specify either the process stack SRO (if the SRO access selector is zero) or a heap SRO (the SRO access selector selects an SRO AD with create rights).

Both CREATE OBJECT and CREATE TYPED OBJECT clear the new object, up to a maximum of 2,048 bytes in the new object's access and data parts. "Clearing" writes zeroes into the data part of the new object and null ADs into the access part of the new object. When the object has been cleared, the completed bit in the object's OD is set. If a new object is larger than 2,048 bytes, then no part of it is cleared, the Clear Memory Size Fault is raised, and the completed bit is cleared. These operators also raise a fault if there is not a large enough block of contiguous physical storage in the specified SRO to allocate the new object. The new AD returned by a successful CREATE OBJECT or CREATE TYPED OBJECT operation has all access rights, delete rights, and has unchecked copy rights if and only if the new object is at level 0.

The new AD returned by CREATE REFINEMENT or CREATE TYPED REFINEMENT has whatever access rights were specified on the AD provided for the base object. The new AD has delete rights and has unchecked copy rights if and only if the new refinement is at level 0.

For all four CREATE operators, an Object Descriptor Exhaustion fault can occur if the specified SRO's object table is full.

An iAPX 432 operating system should handle these fault conditions transparent to user software, automatically expanding object tables and physical storage objects as needed, and clearing large objects. If necessary, user processes can be suspended until the requested storage is available, then restarted transparently.

OBJECT LIFETIME STRATEGIES

A basic characteristic of iAPX 432 memory management is that storage reclamation can be transparent to all programs except the operating system kernel. Programs can create objects but do not need to delete them -- the operating system and the iAPX 432 architecture cooperate to detect when objects are no longer used and then reclaim them. An object's lifetime strategy determines when and how it is deallocated, and derives from the lifetime strategy of the SRO used to create the object. An operating system may also support explicit deallocation of objects by calling an O.S. procedure.

STACK LIFETIMES

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

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 stack SRO is used by the context to create objects local to the context. RETURN deletes all objects local to the context that are created by the context. Note that context objects are preallocated by the operating system and are not dynamically allocated and deallocated using stack SROs.

GLOBAL HEAP LIFETIMES

The least restrictive (and most time-consuming) way to deallocate objects requires an exhaustive search of memory that determines what objects are no longer reachable from the programs in the system via a chain of ADs. Because 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 via a chain of ADs is unusable and can be deleted. Such unreachable objects are called garbage, and the operating system program that finds and reclaims garbage objects is called the garbage collector. The garbage collector can execute as a separate operating system process that is able to run concurrently with other system and user processes. The garbage collector algorithm is based on that of Dijkstra, et al ("On the Fly Garbage Collection: An Exercise in Cooperation," Communications of the ACM, November 1978). The GDP performs one crucial part of the algorithm (the "mutator" role), setting a Copied bit in an object descriptor whenever an AD that references the object is copied. 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 context or process in which they are created.

LOCAL HEAP LIFETIMES

The third lifetime 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 reclaimed 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 (that have not previously been garbage collected) are reclaimed by the operating system.

Using a local heap, a context can create objects that are not local to it, but have the scope of a superordinate, calling context. For example, suppose A calls B, B creates a local heap at B's level, and B then calls C, passing an AD for the new local heap. C, and any other procedures that C calls, can use the local heap SRO to create objects with the same scope as objects created by B using the stack SRO. When B returns, the local heap and objects created from it are reclaimed.

Reclaiming a local heap is not done by the GDP, but must be done by operating system software. The operating system can remove Return Rights from the AD to Calling Context in the context associated with the local heap; this will cause a fault when the RETURN operator is executed, and the operating system can then reclaim the local heap and all objects allocated from it (relatively straightforward because the local heap references a distinct object table used only for objects allocated from it).

FRAGMENTATION AND COMPACTION

Fragmentation is the division of free physical storage into noncontiguous 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 single block of contiguous storage is large enough.

Compaction reduces fragmentation of nonfrozen memory 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. Compaction can be done by the operating system as a parallel process, invisible to users. The GDP supports compaction by providing the Allocated bit in storage descriptors, which determines whether the Base Address field is valid. While an object is being relocated (and temporarily inaccessible), compaction can clear this bit, then move the object, assign the new Base Address, and set the Allocated bit again. (This process is complicated by the need to flush multiple processor caches of address information, as described in Chapter 6, "Processor Management.")

MEMORY MANAGEMENT TRANSITIONS

Figure 4-2 ties together the three lifetime strategies, garbage collection, and compaction in one illustration, showing the transistions between free memory, allocated objects, and garbage.

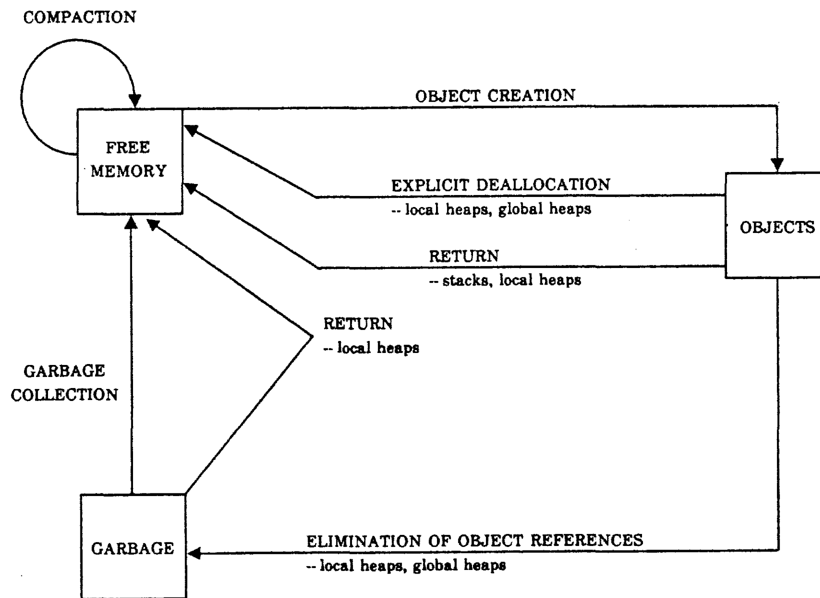


Figure 4-2. Memory Management Transitions

VIRTUAL MEMORY

A computer architecture supports virtual memory if the architecture and operating system can together create the illusion that the system's main memory is larger than the amount of physical main memory in the system. For example, a program in a virtual memory system may occupy 500K bytes of main memory, but at a particular instant, only 200K bytes of that program may be present in the system's physical main memory. The remaining 300K bytes of the program reside on a mass storage device (e.g., a disk). If the program makes a reference to information that is on the disk and not present in physical memory, the operating system intervenes and "swaps", moving the referenced information from the disk to main memory, and possibly moving least-referenced information from main memory to the disk in order to keep enough free space in the main memory.

The intervention of the operating system to implement virtual memory is transparent to most user programs; programs may execute more slowly but their code does not need to be changed. Virtual memory makes a smaller faster memory appear as a larger, slower memory.

Virtual memory provides these important advantages:

1. Programs can run on systems with varying amounts of physical memory without modifying the program to take advantage of the additional memory or to fit in less memory. Programs can be written to run in a large virtual memory, and will execute faster or slower if their is more or less physical memory in a system.

2. Programs can be simplified because the operating system automatically swaps in and out to mass storage as needed, eliminating the need for application programmers to use program overlays and intermediate data files.

3. In a multiuser system, an additional user does not have to be denied access to the system if the system's physical memory is full; it is possible to execute the new user's job and all jobs simply run more slowly. (Note that there is still some upper limit on user activity on the system, at which performance is severely degraded and the I/O channels between main memory and mass storage are saturated.)

In a conventional virtual memory system, the units moved between main physical memory and mass storage are fixed-size pages. For example, a page may be 1K bytes, so that physical addresses 0 to 1,023 are page 0, addresses 1,024 to 2,047 are page 1, etc. A page may be the same size as a disk block, to optimize swapping transfers. In such a system, a page table contains descriptors for the pages that map virtual page numbers to physical page numbers and that contain virtual memory control information, such as whether a page is present in main memory or not.

In a conventional virtual memory system, the unit of protection is large, typically a job, which is mapped into a sequence of contiguous pages. All of these pages have the same protection attributes which isolate the job from other jobs in the system. Each job has a separate address space and to communicate with other jobs or with I/O devices requires an operating system call to copy the information being transferred.

In the iAPX 432, the object addressing mechanism supports both protection and virtual memory. In an iAPX 432 virtual memory system, the units swapped to and from mass storage are objects, not arbitrary pages. (Note: To improve efficiency, an iAPX 432 operating system may group related small objects together into a swapping set that is swapped together, but the set still consists of distinct integral objects, and the grouping is not visible to users.)

All of the iAPX 432 architecture's support for virtual memory is in the object descriptor:

The allocated bit is set if the object is currently allocated in physical memory, and can be cleared by the operating system if the object is swapped out.

The accessed bit is set by the GDP whenever the object is read or written. This bit can be periodically cleared by the operating system; then if the bit is set when looking for objects to swap out, the operating system can leave the object in main memory because it has been recently used.

The altered bit is set by the GDP whenever the object is written. An object that has not been altered may be a good object to delete from main memory if space is needed. This is because such an object can be deleted without swapping it out; the version on disk is still up-to-date.

Because the physical address of an object is contained only in its single object descriptor, swapping an object into a different range of memory addresses than it has previously used is straightforward; only the base address field of its single object descriptor must be changed.

FROZEN MEMORY

In any iAPX 432 system that supports object relocation (compaction or virtual memory), operating system designers need to distinguish memory that is not normally relocated or otherwise made inaccessible, frozen memory. Frozen memory may constitute a distinct part of physical memory, with a distinct frozen global heap SRO; any stack SROs or local heap SROs allocated from the global heap SRO should also be frozen. Frozen memory is used for objects that should never be inaccessible, such as the object table directory, processor objects, and the objects used by the memory management kernel itself. (Note: Memory management may need to expand the object table directory if it becomes full, but this is a special operation that would not be done by the normal compaction process, and that would relocate the OTD within frozen memory itself.) Memory that is not frozen is normal memory.

MULTIPLE PROCESSORS AND MEMORY MANAGEMENT

To improve performance in address translation, iAPX 432 processors cache addressing information for referenced objects and object tables. Whenever an iAPX 432 operating system process modifies addressing information in object descriptors, it must make sure that all processors in the system flush their addressing caches so that they are using correct information. This can be done by broadcasting a REQUALIFY DATA OBJECT CACHE or REQUALIFY OBJECT TABLE CACHE interprocessor message (IPC) to all processors in the system (including to the processor that is executing the operating system process), and waiting for all to respond. The caches and how they are flushed are described in detail in Chapter 6, "Processor Management."

CHAPTER 5
PARALLEL PROCESSING

This chapter describes the iAPX 432 architecture's support for true parallel processing, simultaneous execution of multiple programs by multiple processors. This chapter covers these topics:

- processes as units of parallel execution
- interprocess communication
- support for process and processor synchronization
- transparent multiprocessing
- process scheduling

Chapter 6, "Processor Management," describes GDP caches, interprocessor communication, GDP dispatching modes, and GDP initialization.

PROCESSES

A process represents a program activation or subprogram activation that can execute at the same time (in parallel, concurrently) as other processes. For example, if three persons are using the same computer system at the same time, then each user can be represented by a separate process. Processes can also be used to represent I/O devices. Thus a printer and a card reader attached to a system can operate concurrently if they are handled by separate processes. An iAPX 432 process is represented by a distinct type of system object, called a process object. Also associated with a process and its process object are these other objects:

- process globals object
- process stack object table
- process stack physical storage object
- process carrier
- a current context, one of a doubly-linked list of preallocated contexts for the process

Objects not subordinated to the process but referenced by it include a dispatching port, scheduling port, and fault port.

The process globals object of a process is part of the access environment of every context executing within the process. The process globals object can be entered as an environment by using the COPY PROCESS GLOBALS operator. The process globals object can be used by an operating system, e.g., to reference standard I/O interfaces to be used within the process.

The process stack object table and stack physical storage object are part of the process stack SRO, used for stack allocation and deallocation of objects local to contexts of the process. Storage allocation, deallocation, and stack SROs are described in more detail in Chapter 4, "Memory Management."

The process carrier is the default carrier used in interprocess communication by the process, and is also used for process scheduling and dispatching.

Context objects, preallocated contexts, and their role in program organization are described in Chapter 2, "Program Organization."

A process is sent to its dispatching port to schedule it for execution and then dispatch it to run on a particular GDP.

A process is sent to its scheduling port if it has been allotted a limited number of periods (time slices) executing before its scheduling parameters must be reevaluated by operating system software; the process is sent to the scheduling port when it has used up its allotted number of periods.

A process is sent to its fault port if it encounters a process-level fault. Faults, fault levels, and fault handling are described in Chapter 12, "Fault and Trace Reference."

The data part of a process object contains scheduling parameters, used to decide the order in which competing processes are dispatched to run on a processor; a process data part also contains status information and a process clock that records the execution time received by the process.

INTERPROCESS COMMUNICATION

Many programming applications that use multiple processes require those processes to communicate; for example, if user programs and I/O devices are represented by processes, then printing a report requires transferring information from the process executing the user program to the process handling the printer. Two natural parts of interprocess communication are queuing and blocking. Suppose that one process in a system manages all disk transactions, reading and writing blocks on behalf of requestor processes and then sending them acknowledgement and any results (such as the value of a read-in block). If requests for disk transactions arrive while the disk process is busy, the new requests must be queued, must wait. A process that makes a disk request must usually wait for the request to be processed before it can execute further; i.e., it blocks waiting for the reply. Similarly, when no process is requesting disk transactions and all previous work is done, the disk service process blocks waiting for new work. When a process blocks, the GDP executing it is freed to execute some other process that is ready to run.

These objects are used in interprocess communication:

- messages any iAPX 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.

The iAPX 432 provides both simple and enhanced forms of interprocess communication; the simple model is the most frequently used.

In the simple model, processes are the active agents that send and receive messages at ports, and the process carriers are implicitly used. 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 that transfer a message only if the operation does not block. A Boolean parameter is assigned true or false to indicate the success or failure of such a conditional operation.

In the enhanced model, surrogates can be created that 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 (although the queue of waiting processes and 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.

The simple and enhanced model 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 5-1 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 ADs, just as if they were explicitly copied. Note that null ADs containing embedded data values can be used to transfer small messages (ordinal values in the range $0 .. 2^{31} - 1$) without referencing a message object. Embedded data values are described in Chapter 9, "Object Set."

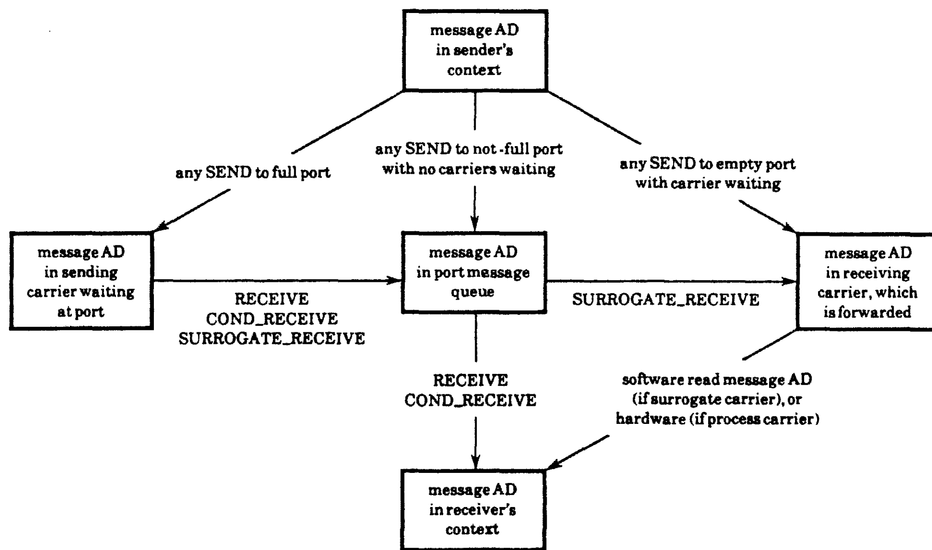


Figure 5-1. Message AD Transfer

PORTS

Ports are system objects that provide queuing mechanisms supported by the iAPX 432 processors, consisting of two queues for each port, a bounded message queue and an unbounded carrier queue. The message queue of a port contains the message ADs 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 message queue. The queuing value is either 0 (for simple port operations or FIFO ports) or is determined from the surrogate carrier used in sending the message. The port's queuing discipline specifies that messages are enqueued either FIFO, by priority, deadline within priority, or by delay. Deadline within priority enqueuing is normally used only for dispatching ports. Delay enqueuing is normally used only for delay ports. Both dispatching and delay ports are described in subsequent sections of this chapter. 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.

Ports are completely described in Chapter 9, "Object Set."

CARRIERS

Carriers transport messages to and from ports. Also, surrogate carriers provide message queuing values. A process carrier is associated with each process. Process carriers are used by the SEND and RECEIVE operators. If a process carrier blocks at a port, then the associated process blocks also. The forwarding of a process carrier sends it to a dispatching port so that its associated process can run.

Surrogate carriers act as surrogates on behalf of processes. For example, a process can wait at multiple ports for any message sent to any of the ports, by creating a surrogate to wait in its place at each of the ports. Each of the surrogates is forwarded to a single common port whenever it receives a message. The process can then wait at the single common port for the arrival of a surrogate, remove the message received by the surrogate, and cause the surrogate to again wait at its assigned port for another message.

Processor carriers are a third type of carrier, but are not used in interprocess communication. All three types of carrier are described in Chapter 9, "Object Set."

SENDING MESSAGES

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 first port, the carrier is forwarded to the second port.

Two operators 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, then the message is copied into the sending carrier, which is appended to the FIFO queue of waiting carriers.

2. Otherwise, if the port is empty and carriers are waiting to receive messages, then the message is copied into the first waiting carrier, which is removed from the carrier queue and forwarded to its second port.

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

A SEND 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 nonblocking SEND does not involve the process carrier. In SURROGATE SEND, the surrogate carrier is forwarded to its second port even if it does not block at the first port.

The queuing value used to insert the message is zero for SEND or if the port's queuing discipline is FIFO. For a SURROGATE SEND to a priority port, the queuing value is the priority from the surrogate carrier.

A third operator, CONDITIONAL SEND, never blocks. If the port is full, the message is not sent and a Boolean destination operand is cleared to false. Otherwise, the message is sent as described for SEND and the boolean parameter is set to true.

RECEIVING MESSAGES

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 operators 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 operators:

1. If the port is empty, then 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 the port is full and carriers are waiting to send messages, then 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, and must then read the carried message from the surrogate carrier.

A third operator, CONDITIONAL RECEIVE, never blocks. If the port is empty, no message is received and a Boolean destination operand is cleared to false. Otherwise, the message is received as described by RECEIVE and the Boolean parameter is set to true.

FORWARDING CARRIERS

When a SURROGATE SEND or SURROGATE RECEIVE operation 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 optional; if the second port AD is a surrogate operation is null, the carrier is not forwarded. A forwarded carrier is itself the message that is sent to the second port. A carrier forwarded from a SURROGRATE SEND operation carries no other message. A carrier forwarded from a SURROGATE RECEIVE operation carries not only itself but the received message as well; such a carrier must be received as a message from its second port, before reading the carried message.

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 AD from the process carrier to the receiving context (and then nulling the message AD in the process carrier). A difference in forwarding process carriers and surrogate carriers is that surrogate carriers are always forwarded (if their second port is not null) while process carriers are forwarded only if an 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.

PROCESS AND PROCESSOR SYNCHRONIZATION

Any system that supports multiple processes must provide means to synchronize the execution of processes that share data or resources, so that they do not interfere with one another. For example, if several processes use a shared printer for their reports, the processes must synchronize use of the printer. Otherwise, a line sent from process A could be followed by a line sent from process B, etc., making the output unreadable.

A process may need exclusive access to a particular data structure or resource for a very short time or for a much longer time. For example, a process that is updating a shared field by adding 5 to the present value needs exclusive access to the field for the time required to read the old value, add 5, and write the updated value--a duration of a few microseconds or less. For another example, a process that is printing a lengthy report may need exclusive access to the printer for many minutes. An example that is between these two in duration is the searching and updating of shared object data structures in certain high-level GDP operations, such as sending a message to a port or creating a new object from a shared heap SRO; these operations can take dozens of microseconds.

Three synchronization strategies are used in the iAPX 432: indivisible operators, object locks, and using interprocess communication for synchronization.

Indivisible operators are used to update a single short-ordinal or ordinal data field in a single indivisible uninterruptable transaction. A special Read-Modify-Write (RMW) bus cycle is used by these operators, so that no other processor can access such a field in the interval between reading and writing it. Indivisible variants of the short-ordinal and ordinal ADD and INSERT operators are provided. The previous value of the modified field is pushed onto the operand stack by these operators.

An object lock is a double-byte field in the data part of an object, used to get and release exclusive access to the object. Many system objects contain object locks. GDP operators, such as SEND or CREATE OBJECT, always use object locks in such objects as ports and heap SROs to ensure exclusive access; operating system modules that manage such system objects should also respect the locking conventions, which are defined in Chapter 9, "Object Set." Object locks are used by processors (e.g., when dispatching from a dispatching port), by processes for a single instruction (such as a SEND operation), and by processes for multiple instructions, delimited by the operators LOCK OBJECT and UNLOCK OBJECT. The LOCK OBJECT operator, if it finds that the lock is not available, waits 300 processor cycles and retries the lock, waiting and retrying up to 32 times before returning an indication that the lock operation failed. Object locks are appropriate and efficient for synchronization when a lock is normally held for a short time (i.e., less than a millisecond). An advantage of object locks is that the identity of the locking processor or process is stored in a busy lock, and only the same processor or process can normally release the lock.

Ports and interprocess communication can be used for process synchronization in two ways: by using a server process and by using a port as a semaphore. In the server process approach, the resource or data structure to be synchronized is only accessed by a single server process. For example, a disk drive may be accessed by only a disk server process. Other processes that need to access the disk send request messages to a request port; the disk server process receives the requests, acts on them one at a time, and acknowledges each request by sending a reply message to a reply port; each requestor normally uses a separate reply port, and the reply port to be used for a particular request can be specified in the request message.

In the semaphore approach, a port with one entry in its message queue is used to indicate if the resource or data structure is available or not. The entry itself is not relevant, but simply indicates resource availability by its presence or absence. For example, the port being empty can indicate that the resource is available. A process gets exclusive access to the resource by SENDing a message to the port. The port is now full and subsequent SENDs by other processes trying to obtain the resource will block. When the using process no longer needs exclusive access to the resource, it RECEIVEs from the port. If no other processes are waiting, the RECEIVE makes the port empty and indicates that the resource is available; if processes are waiting, then the first waiting process unblocks and has exclusive access to the resource.

One advantage of the server process approach is that, properly designed, service cannot be disrupted by the abnormal termination or suspension of a requesting process; if a requesting process fails, it simply never picks up its reply from its reply port. (Note that the server process should use the CONDITIONAL SEND operator and not SEND to send the reply to the reply port, so that the server process never blocks because of a problem with the requestor-specified reply port; it is up to the requestor to ensure that the reply port will not be full and the CONDITIONAL SEND will succeed.) In contrast, when using a semaphore, if a process is abnormally terminated while holding the semaphore, then all other users of the semaphore will block when they attempt to get it.

TRANSPARENT MULTIPROCESSING

iAPX 432 programs can be designed and implemented independent of the number of GDPs in a system. GDPs are homogeneous servers that execute ready processes, and more or fewer GDPs simply means faster or slower execution, without causing software changes. This is called transparent multiprocessing. A good analogy is a modern bank, with a single line of customers served by (a varying number of) multiple tellers. The transactions and how they are carried out (the programs) are independent of which teller waits on a customer; for the purpose of doing normal business with the banks, the tellers are interchangable. A change in the number of tellers does not change how the bank does business, but simply makes customer service faster or slower.

Transparent multiprocessing is implemented by the GDP architecture using the same basic mechanisms as interprocess communication: ports, carriers, and messages. A special port, called a dispatching port, is used to queue ready processes that are waiting for a GDP to execute them. Dispatching ports normally use a more complex queuing discipline than other ports, deadline within priority queuing; this queuing discipline is designed to support process scheduling. Processes are forwarded to dispatching ports using their process carriers. GDPs pick up processes to run by receiving them from dispatching ports, using their processor carriers. The messages being sent and received are the processes themselves. If a dispatching port is empty, then there are no processes ready to run at the port, and a GDP attempting to receive a process itself blocks, idling. When another GDP sends a process to the dispatching port, it "wakes up" the waiting GDP to execute the process; the waking up uses the interprocessor communication facilities described in Chapter 6, "Processor Management."

PROCESS SCHEDULING

If there are more processes ready to run than there are GDPs in a system, then processes must compete for execution time, to determine which processes will execute before other processes, and to determine how long a process can execute before it must give another process a turn. Determining the order in which ready processes will be dispatched and how long they can run is called process scheduling. Four parameters control process scheduling: priority, deadline, service period, and period count. Priority and deadline are contained in the process carrier; service period and period count are contained in the process object. The deadline and service period parameters use the concept of a system time unit (256 microseconds in 432/600 systems), the GDP's basic "tick" or unit of passing time, determined by an external circuit that periodically asserts the GDP's PCLK pin.

The priority field is a short ordinal; the lowest priority is zero; the highest is 65,535. When processes with different priorities are at a dispatching port, the process with the highest parameter is always dispatched first.

The deadline field is a short ordinal ranging from 0 to $2^{14}-1$ that indicates the number of system time units that a process can or should wait for dispatching relative to other processes. When a process is queued at a dispatching port, its deadline value is equal to the number of system time units that it has waited in the message queue minus the value from its deadline field. For example, consider process A that has waited 400 time units for dispatching, with a deadline parameter of 500, and process B that has waited 100 time units for dispatching, with a deadline parameter of 50. A's deadline value for queuing is 400 - 500 = -100. B's deadline value for queuing is 100 - 50 = 50. B is queued ahead of A, even though A has waited longer (presuming A and B have the same priority).

Processes in a dispatching port's message queue are ordered first by priority, then by computed deadline value (computed as described above), and last, FIFO within the same deadline value within the same priority value. Note that if a dispatching port is full, so that ready processes are in the port's carrier queue, that the processes in the carrier queue are in FIFO order, and only scheduled when inserted in the message queue.

A process's service period is the number of system time units that the process is allowed to execute before being suspended and redispatched, to allow other processes to compete for execution time.

The period count parameter is the number of times that the process can be dispatched before being sent to its scheduling port. It can be used by operating system software to impose a time limit on total process execution time, or to periodically "tune" process scheduling parameters. The period count is a short ordinal; a period count value of 65,535 indicates an infinite period count which is never decremented and never causes the process to be sent to its scheduling port. For a finite period count N, the process's execution time is limited to N times its service period, before the process is sent to its scheduling port. If operating system software needs to halt execution of a process, it can set the period count to zero, guaranteeing that the process will report to the operating system via its scheduling port as soon as any current service period is completed.

CHAPTER 6
PROCESSOR MANAGEMENT

This chapter describes these aspects of processor management:

1. GDP caches
2. interprocessor communication
3. normal GDP execution cycle
4. GDP dispatching modes
5. GDP initialization

GDP CACHES

This section describes information cached by GDPs from various objects to support object addressing and program execution. Internal GDP registers or associative memory holds copies of frequently used information, such as the current instruction pointer or the physical address of the most-recently-used object table. These internal memories (caches) cannot be directly read or written by iAPX 432 programs, which can only access information within objects.

The caches significantly speed up program execution. The only programmers who need to concern themselves with the caches are operating system designers and implementers. The caches are of concern to operating system programmers only when the caches hold different values than the fields that they should represent in different objects. Ensuring cache integrity is complicated by multiprocessing, in which multiple GDPs may cache the same fields. There are five GDP caches: data object cache, object table cache, context cache, process cache, and processor cache.

DATA OBJECT CACHE

All iAPX 432 instruction operands are located in the data part of some object; even operations on access descriptors must designate the ADs via an indirect access selector in the data part of some object. The data object cache contains addressing information, taken from object descriptors, for the most-recently-used "data objects" (objects with an operand in their data part). Note that no information from these objects is cached, but just addressing information (such as base address and length). The cache is associative; every time a GDP references an operand in the data part of an object, the cache responds (a "hit") if addressing information for that object is in the cache. Otherwise the cache indicates a "miss"; the addressing information is not in the cache and must be read from the object descriptor in memory; the information read is then added to the cache, displacing the least-recently-used entry.

There are some operating system actions that can invalidate entries in the data object cache. The following two examples illustrate such actions and how the operating system can maintain cache integrity:

1. If a data object is to be relocated in memory (by compaction) or swapped out to disk (in a virtual memory system), then it must be marked as inaccessible, by clearing the Allocated bit in the object's OD. The operating system software that clears this bit in the OD must ensure that all processors (GDPs and IPs) update their data object caches before the object is relocated or swapped. Otherwise, a processor with cached addressing information for the object could read or write memory where the object used to be, a protection violation. To ensure that all data object caches are updated, operating system software must send the REQUALIFY DATA OBJECT CACHE IPC to all processors, and wait for all processors to acknowledge that they have received and executed the IPC.

2. If an object is to be deallocated while ADs for it may still exist (i.e., before it is eligible for garbage collection or deallocation by Return from a subprogram call), then the object's OD must be marked as invalid before deallocating memory used by the object. The operating system software that changes the OD must ensure that all processors (GDPs and IPs) update their data object caches before the object's memory is deallocated. Otherwise, a processor with cached addressing information for the object could read or write memory where the object used to be, a protection violation. To ensure that all data object caches are updated, operating system software must send the REQUALIFY DATA OBJECT CACHE IPC to all processors, and wait for all processors to acknowledge that they have received and executed the IPC.

OBJECT TABLE CACHE

The object table cache contains addressing information, taken from object descriptors, for the object tables most-recently-used for object addressing. Note that this cache doesn't contain information from the object tables but from the ODs for the object tables. The cache is associative. Every time a GDP references an operand, it is via some object table; the cache responds (a "hit") if addressing information for the object table is in the cache. Otherwise, the cache indicates a "miss"; the addressing information must be read from the object table OD in the Object Table Directory in memory; the addressing information read is then added to the cache, displacing the least-recently-used entry.

If an object table is to be relocated in memory (by compaction) or swapped out to disk (in a virtual memory system), then it must be marked as inaccessible, by clearing the Allocated bit in the object table's OD. The operating system software that clears this bit must ensure that all processors (GDPs and IPs) update their object table caches and data object caches before the object table is relocated or swapped. The data object caches must be flushed in case any of the data object's ODs come from the now-inaccessible object table; all such objects must now be inaccessible as well; if the addressing information for such objects is no longer cached and the addressing information for the object table is no longer cached, then any attempt to reference such objects will fault because the object table is inaccessible. To ensure that all these caches are updated, operating system software must send the REQUALIFY OBJECT TABLE CACHE IPC to all processors and wait for all processors to acknowledge that they have received and executed the IPC. This IPC flushes both the object table cache and the data object cache in each processor that receives and executes it.

CONTEXT CACHE

These fields in the current context are cached by the GDP:

- AD to Current Context (Environment 0)
- AD to Environment 1
- AD to Environment 2
- AD to Environment 3
- Context Status
- Operand Stack Pointer
- Instruction Object DAI
- Instruction Pointer
- top double-byte (if any) of the Operand Stack

All of these fields except the AD to Current Context can change within the GDP during context execution, without being updated in memory. Thus, these fields cannot be read from memory or altered by writing into memory during context execution. The Current Context AD can be read, but lacks delete rights and cannot be written.

The Environment 1, 2, and 3 ADs in the GDP are modified by ENTER ENVIRONMENT or COPY PROCESS GLOBALS operators. The Context Status in the GDP is modified by the SET CONTEXT MODE operator. The Operand Stack Pointer in the GDP is modified by stack addressing modes or the ADJUST STACK POINTER operator. The Instruction Object DAI and Instruction Pointer in the GDP are modified by branch instructions, context fault-handling, or trace-handling. The top-of-stack register is filled when a double-byte is pushed on the stack; any double-byte value already in the register is flushed to memory. The ADJUST STACK POINTER operation flushes any value in the top-of-stack register to memory.

When a context is called, all cached values for the calling context are flushed to memory, except the Current Context AD, which cannot have changed within the calling context. Then all context cache entries are filled with their initial values for the new context. The Environment 2 and 3 ADs are always null initially. The Context Status is inherited from the calling context and thus need not be reloaded. The top-of-stack register is initially marked as empty.

When a context returns, there is no need to update memory with cached fields, because all cached fields are local to a call, and the call is over. On return, all the cache entries are reloaded from the returned-to context (except the top-of-stack register, which is just marked as empty). Note that the context status is reloaded, so that any changes in context status made in the called context are local to the call.

The REQUALIFY CONTEXT IPC flushes the context cache and the data object cache of the processor that receives and executes it.

PROCESS CACHE

The process cache contains information cached by the GDP from a process executing on the GDP. This section does not list the fields cached. However, the field Period Count is guaranteed not to be cached. This allows operating system software to force a process to report to its scheduling port, by writing a zero in this field.

When a process stops executing on a processor (to wait at some port), the process cache and context cache are flushed to memory. When a processor dispatches a process, the process cache and context cache are loaded from memory. The data object cache and object table cache are not affected in any special way by process suspension or processor redispatching.

A process (at any instant) can be executing on only one GDP, therefore, only one GDP can be caching information in its process cache for a particular process at a particular instant.

The REQUALIFY PROCESS IPC flushes the process cache, the context cache, and the data object cache of the processor that receives and executes it.

PROCESSOR CACHE

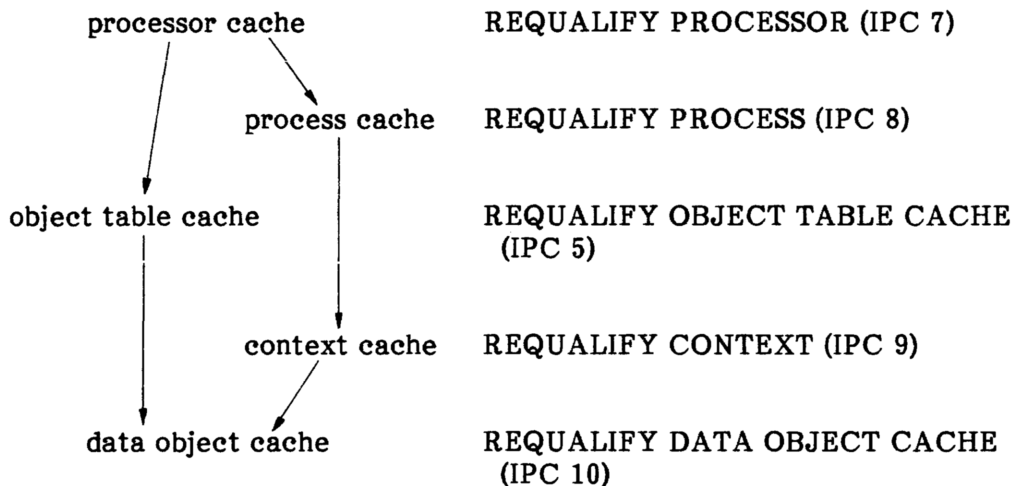
The processor cache contains information cached by the GDP from its processor object. This section does not list the fields cached. However, the processor cache does include addressing information for the Object Table Directory in addition to fields from the processor object. The REQUALIFY PROCESSOR IPC flushes the processor cache and all other GDP caches. For example, all processors would be sent this IPC if operating system software had to expand and relocate the Object Table Directory in memory.

NOTE

The REQUALIFY PROCESS IPC does not function correctly on current GDPs if the GDPs are allowed to become idle. The operating system designer should either provide special "idle processes" to keep any GDPs from going idle, or should use the REQUALIFY PROCESSOR IPC in place of the REQUALIFY PROCESS IPC.

CACHE SUMMARY

Figure 6-1 shows the GDP caches, the relationships between the caches, and the IPCs that flush various caches.



NOTES

Flushing the data object cache does only that.
Flushing the object table cache also flushes the data object cache.
Flushing the context cache also flushes the data object cache.
Flushing the process cache also flushes the context cache and the data object cache.
Flushing the processor cache flushes all caches.

Figure 6-1. GDP Caches

INTERPROCESSOR COMMUNICATION

Processes can affect the operation of processors by sending them InterProcessor Communications (IPCs) using the SEND TO PROCESSOR operator. Interprocessor communication to a processor is via one of two Processor Communication Objects (PCOs) referenced by the processor object. The local PCO, unique to the processor, is used to send an IPC to just one processor. The global PCO, shared by all processors or a pool of processors, is used to send an IPC to all processors that reference the global PCO. Chapter 9, "Object Set," describes Processor Communication Objects. Chapter 10, "Operator Set," describes the SEND TO PROCESSOR operator. Table 6-1 lists IPCs defined for GDPs.

Table 6-1. GDP IPCs

Code	Category/Name	Notes
	Control:	
0	Wakeup	Used in dispatching, to wake up an idle processor when a process is bound to it.
1	Start	Causes a processor to begin executing a process bound to it or to proceed to a dispatching port, depending on its state and current dispatching port.
2	Stop	Causes a process to stop executing any current process, flush its process, context, and data object caches, and idle, waiting for another IPC.
6	Reset Processor	Causes a processor to flush all caches and execute the processor initialization sequence.
	Global IPC Acceptance:	
3	Accept Global IPCs	
4	Ignore Global IPCs	
	Cache and Object Requalification:	
5	Requalify Object Table Cache	
7	Requalify Processor	
8	Requalify Process	
9	Requalify Context	
10	Requalify Data Object Cache	
	Dispatching Modes:	
11	Enter Normal Mode	
12	Enter Alarm Mode	
13	Enter Reconfiguration Mode	
14	Enter Diagnostic Mode	

NORMAL GDP EXECUTION CYCLE

When a GDP is executing a process, it executes instructions and checks for three different events between instructions:

- receipt of an IPC, via local or global PCOs (global only if processor is set to accept global IPCs). The IPC is executed and acknowledged. Control then resumes, unless the IPC changed the dispatching mode or otherwise suspended process execution.
- process timeout; the process must be suspended and redispatched (if periods remaining) or sent to its scheduling port (if no periods remaining); the processor must then redispatch.
- a trace event; return information is saved and the next instruction will be taken from the trace instruction object as described by Chapter 12, "Fault and Trace Reference."

The order in which these different events are checked for and handled is not defined by this manual.

GDP DISPATCHING MODES

Normal GDP scheduling and dispatching is non-preemptive; once a process is dispatched, it runs until it blocks at some port or until its time slice expires. Even the arrival of a higher-priority process at a dispatching port does not preempt a running process. The lack of preemptive scheduling is not normally a problem in iAPX 432 central systems, because I/O interrupts and real-time processing are handled in peripheral subsystems. However, some exceptional events can require rapid response even in the central system:

1. A power-failure alarm requires orderly system shutdown in fractions of a second.
2. A hardware failure requires immediate system software action to reconfigure the system without the failed component.
3. A violation of system integrity has been discovered (e.g., a corrupted object table or processor object), requiring immediate diagnosis and possibly repair.

It would be wasteful to require a dedicated GDP reserved for each such class of exceptions; instead GDPs can function in one of four dispatching modes. A GDP that receives an IPC to enter another dispatching mode stops executing any current process (flushing caches but <u>not</u> redispatching the process) and can immediately begin executing an alternate process bound to an alternate process carrier. Each GDP can reference four process carriers and four dispatching ports, a process carrier and dispatching port for each dispatching mode.

When entering a new dispatching mode, a processor does the following:

```
IF the new processor carrier is bound to a process,
  begin executing that process.
ELSIF the new processor carrier is queued at an empty dispatching port,
  the processor idles.
ELSE
  the processor redispatches at the new dispatching port.
END IF
```

The four dispatching modes are:

NORMAL — The normal mode for both user and system processes, and the mode in which GDPs start. The ENTER NORMAL MODE IPC is used to return to this mode.

DIAGNOSTIC — Entered by the GDP to handle a processor-level fault, or in response to a master/checker error (hardware HERR pin is also asserted) or if it receives the ENTER DIAGNOSTIC MODE IPC.

RECONFIGURATION — Entered if the GDP receives the ENTER RECONFIGURATION MODE IPC, and intended to be used for dynamic hardware reconfiguration, especially in fault-tolerant systems.

ALARM — Entered if the GDP's hardware ALARM pin is asserted, or if it receives the ENTER ALARM MODE IPC.

GDP INITIALIZATION

A GDP initializes when its hardware INIT pin is asserted or when it receives the RESET PROCESSOR IPC. The GDP reads its own 8-bit processor ID from interconnect address space location 0 (implemented separately for each processor). The GDP assumes that the initial Object Table Directory begins at physical location 8 in the storage address space. The GDP reads the addressing information for the Processor Object Table from the initial OTD, then uses its processor ID to index into the Processor Object Table and read addressing information for its own GDP processor object. The GDP verifies ("qualifies") the type of the processor object and may check other attributes as well. Initialization does not start the GDP executing any process and does not even queue it at any dispatching port; a START IPC must be sent to the GDP for it to begin dispatching.

CHAPTER 7
INSTRUCTION INTERFACE

Like traditional data processors, the GDP executes a sequence of instructions to accomplish a programmed operation. The GDP's instruction interface is the set of architectural features that define how operators and operand references are encoded to make up instructions, how instructions are fetched and interpreted by the processor, and how operands are addressed via the available operand reference modes. This chapter describes the GDP's instruction interface. The function of instruction components and the modes of operand addressing are described in detail. Basic instruction execution and physical address generation are also described.

INSTRUCTION EXECUTION ENVIRONMENT

CURRENT CONTEXT

The GDP's instruction interface is fundamentally affected by the fact that the iAPX 432 architecture is object-based. This is reflected in the fact that the GDP executes user programs that are encoded in system-typed objects called instruction objects. These instruction objects are part of the defining domain of the current program environment. The domain object is the architecture's object representation of the Ada language's package construct.

The currently active execution environment is represented by another system-typed object called a context object. The current context is best thought of as the activation record for an invocation of a subprogram or procedure (programmed, for example, at the source level within an Ada package). Among other things, the current context defines the logical access environment currently available to the program.

The context (or logical) access environment consists of four environments. The current context object is one of them and is not changeable. The other three are dynamically changeable under program control. The three dynamic environments are referred to as ENV1, ENV2, and ENV3. The current context itself is called Environment 0 (ENV0). Each entered environment object contains ADs in its access part that reference objects directly accessible to the program via that environment. Since a running program can dynamically change its own logical access environment by using appropriate operators in the operator set, the four environments together define the instantaneous access environment of the program. Since the instantaneous access environment can only be changed by explicit instructions, it does not change during operand evaluation. For more information on access environments see the Program Organization chapter of this manual.

Figure 7-1 illustrates the general instruction execution environment.

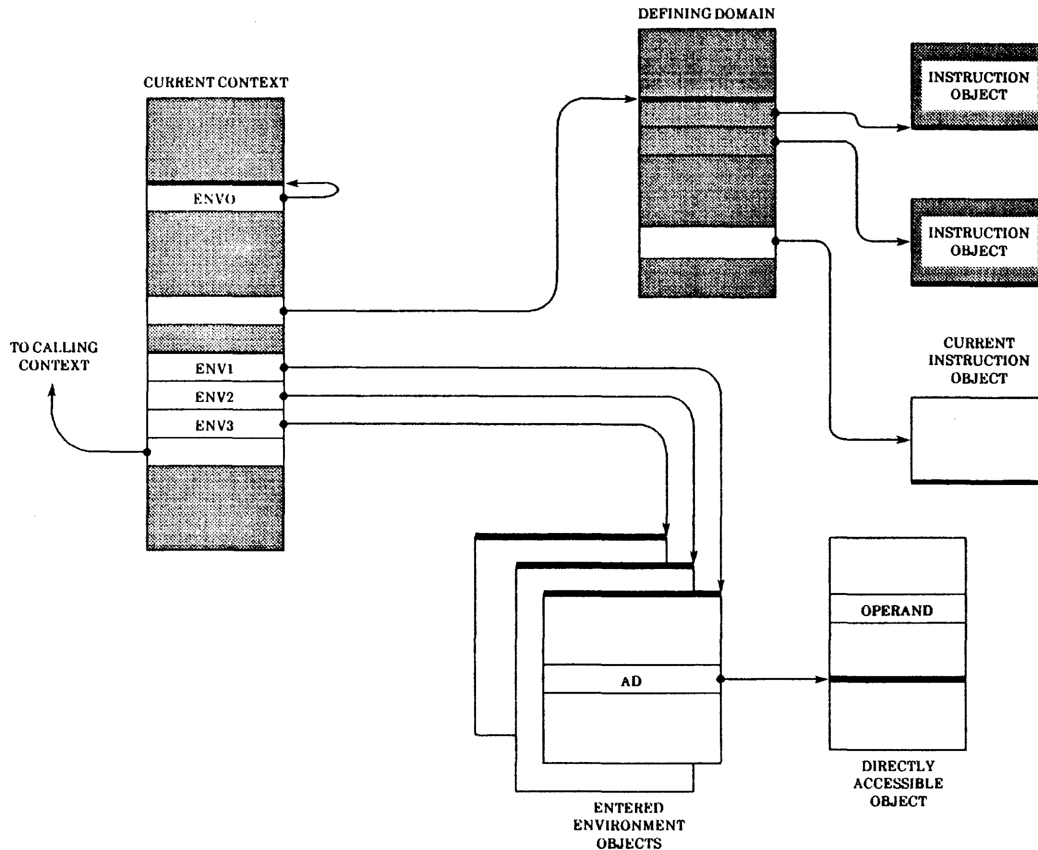


Figure 7-1. Instruction Execution Environment

INSTRUCTION OBJECTS

Instruction objects are system-typed objects containing a sequence of intructions that constitute a programmed software operation. Typical compilers will compile the instructions for a source-level subprogram (procedure or function) into an instruction object. Only instructions obtained from such a system object are executable. Any attempt to execute instructions from some other type of object will cause the GDP to fault.

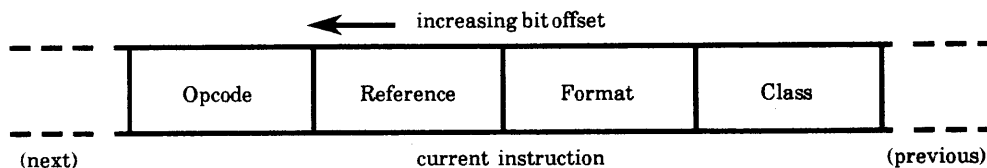
Unlike data items, instructions are not constrained to fit within the fixed length formats characteristic of the computational data types. Instead, a GDP views an instruction object as containing a contiguous stream of bits called an instruction stream. Individual processor instructions, which contain a variable number of bits, may begin at any bit displacement within an instruction object.

The location of a GDP instruction is specified by a bit displacement from the beginning of the instruction object data part to the first bit of the instruction. Such a displacement is limited to a 16-bit representation and thus has a maximum value of 65,535 as a bit displacement, or 8,191 as a byte displacement. Therefore, only the first 8,192 bytes are addressable as instructions.

Regardless of individual instruction boundaries, in its current implementation, the GDP fetches 32-bit portions of the instruction stream at a time for decoding. Thus, the instruction object size must be rounded up to the next 16-bit boundary plus a 4-byte pad. Otherwise, an Instruction Object Displacement fault can occur when the processor attempts to fetch the bits of the last instruction in the object.

INSTRUCTION STREAM

Instructions are variable-length sections of a bit-addressed stream in an instruction object. The GDP interprets these instructions as being composed of fields of varying numbers of bits. The bit stream is scanned from lower-address to higher-address bits. Each field ends when the GDP recognizes a valid encoding; (one valid encoding is never the same as the beginning of another valid encoding;) the next bit in the stream begins the next field or next instruction. The fields are organized to present information to the processor in the sequence required for decoding. Every instruction contains an operator specification and possibly several references. The operator specifies to the processor what operation is to be performed, and the references specify the operands to be used or manipulated. The major fields of an instruction are ordered as follows:



F-0012

Instruction Fields

Class and Opcode Fields

The operator specified in an instruction is encoded in two fields, the Class field and the Opcode field. The Class field specifies the operator class to which the operator belongs, and the Opcode field selects the operator to be performed from within that class. The Class field is always present; the Opcode field is omitted if there is only one operator in the class. The operator's class determines the order of the operator (i.e., the number of operands) and the length of each operand. GDP instructions manipulate zero, one, two, or three operands of varying sizes as specified by the Class field in the instruction.

Format Field

If the Class field indicates one or more operands, a Format field is required to specify whether the references are implicit or explicit and to specify the mapping of data references to operands. The Format field encoding additionally determines which data references--in sequential order--specify which operands. The Format field indicates for each operand whether it is:

- Implicitly accessed at the top of the operand stack, or
- Explicitly specified by a Data Reference field in the instruction.

Operands cannot be specified as literals in the instruction stream; they must always be located in the data part of an object or on top of the operand stack. Note that branch references can be specified literally (directly) in the instruction; but, as such, they are not operands. Branch references are discussed later in this chapter.

The Format field permits the GDP to appear to the programmer as a zero-, one-, two-, or three-address architecture. The order-zero instructions do not require any references and, as a result, do not have a Format field. The order-three Format field encodings allow either of the two source operands to come from the top of the operand stack if both source operands are specified to come from the stack. Thus, the ordering of operands on the stack does not restrict the use of the noncommutative, order-three operators. See the Operand Stack Interaction section of this chapter for more information on operand ordering on the stack.

Redundant operand references, such as those that might occur when a source and destination address are identical, may be specified (using the Format field) in a manner that eliminates the need for their common reference to appear more than once in the instruction. Table 7-1 shows how the format field encodings determine the mappings from the possible data or stack references to their associated operands.

The Format field provides information allowing a single explicit data reference to play more than one role during the execution of the instruction. As an example, consider an instruction to increment the value of an integer in memory (INC_I I,I). This instruction contains, in sequential order:

- A Class field, whose value (1100) specifies that the operator is of order-two and that the two operands both occupy a word of storage
- A Format field, whose value (10) indicates that a single data reference specifies a logical address to be used both for fetching the source operand and for storing the result
- An explicit Data Reference field (whose encoding depends on the operand reference mode) specifying the integer operand to be incremented
- An Op-code field (0001) for the order-two operator INC_I

Table 7-1. Format Field Encodings

ORDER	OPERAND 1	OPERAND 2	OPERAND 3	EXPLICIT REFERENCES	FORMAT ENCODING
0				0	none
1	dref1			1	0
	stk			0	1
2	dref1	dref2		2	00
	dref1	dref1		1	10
	dref1	stk		1	01
	stk	dref1		1	011
	stk	stk		0	111
3	dref1	dref2	dref3	3	0000
	dref1	dref2	dref2	2	1000
	dref1	dref2	dref1	2	0100
	dref1	dref2	stk	2	1100
	dref1	stk	dref2	2	0010
	stk	dref1	dref2	2	1110
	dref1	stk	dref1	1	1010
	stk	dref1	dref1	1	0001
	dref1	stk	stk	1	0110
	stk	dref1	stk	1	1001
	stk1	stk2	dref1	1	0111
	stk2	stk1	dref1	1	0101
	stk1	stk2	stk	0	1011
	stk2	stk1	stk	0	1101
	dref2	dref1	dref3	3	0011
	dref2	dref1	stk	2	1111

NOTES:

dref1,dref2,dref3
indicate that the operand is referenced through the first, second, or third explicit data reference in the instruction's reference field.

stk
indicates that the operand itself is to be pushed onto, or popped from, the operand stack.

stk1,stk2
indicate that the operand is popped from the top (stk1) or next-to-top (stk2) of the operand stack.

Reference Field

The Reference field consists of a sequence of 0 to 3 Data References as specified by the Format field. A data reference is required for explicit specification of an operand location. For branch operators, the Reference field can contain a single branch reference or a combination of a data reference followed by a branch reference. A branch reference in the Reference field always follows any data references that might also be in the reference field.

Frequency Encoding

The Class, Format, and Opcode encodings have been chosen on the basis of the frequency of usage of the operators or modes they encode. Often used encodings are encoded with fewer bits. This reduces both instruction size and execution time for the more frequently used operators and operand reference modes.

Complete composition and encoding information for the instruction fields is presented in Part Two of this manual. See the Operator Set and Instruction Encoding chapters.

OPERAND ADDRESSING

OPERAND TYPES

An operand is one of up to three data items that are defined for an operator. Depending on its class, each operator has a set of operand types defined for it. Most GDP operators require simple computational data types as operands. However, some operand types are not considered data types in the strict sense due to the lack of operations defined for them. These operand types are required by certain operators to fully specify the operation. For example, access selectors are required by many object operators to specify the objects to be used. Yet, the operator set contains no operators that exclusively deal with access selectors as data types. The following operand types are used by the GDP operator set:

- Computational Data Types. These are: Character, Short Ordinal, Short Integer, Ordinal, Integer, Short Real, Real, Temporary Real.

- Boolean. A Boolean is a value of type character used to represent logical TRUE or FALSE.

- Bit Field Specifier. Bit field specifiers specify a field of bits to be manipulated within an ordinal or short-ordinal operand by a bit-field operator.

- Access Selector. Access selectors select an access descriptor in the entered access environments of the current context. They are often required by object operators to specify an object or AD to be used by the operation.

- Domain Access Index. A domain access index selects an access descriptor in the defining domain of the current context.

- Packed Operands. For many object operators (i.e., operators that perform operations on objects as entities), a given operand can be a "packed operand". A packed operand is composed of sub-operands that are instances of the other recognized operand types. For example, operand 2 of the INSPECT OBJECT operator is a packed operand comprised of two 16-bit sub-operands. The least-significant 16 bits contains an access selector for a destination data object and the most-significant 16 bits contains a short-ordinal displacement into the data part of the selected object.

See the Operator Set chapter in Part Two of this manual for more details on operand types.

OPERAND ALIGNMENT

When a GDP executes instructions on behalf of a context, it manipulates operands found within the access environment of that context. An individual operand may occupy one, two, four, eight, or ten bytes of memory. All operands are referenced by a logical address.

The offset component in an operand's logical operand address specifies the number of bytes from the base of an object to the first byte of the operand in the data part of the object. For operands consisting of multiple bytes, the address locates the low-order byte. The higher-order bytes are found at the next higher consecutive addresses.

As an operand, each computational data type can be aligned on an arbitrary byte boundary in the data part of the object, although multi-byte operands may be processed more efficiently when aligned on double-byte boundaries (if the memory system is organized in units of double bytes). Note that Bit 0 of each byte is the low-order bit of that byte. Also note that byte addresses are numbered consecutively. By convention, this manual shows memory addresses increasing from right to left and from bottom to top of the page.

LOGICAL ADDRESS COMPONENTS

In an iAPX 432 system, all processors (of any type) can access the contents of all of the available physical memory. All iAPX 432 processors access memory via a two-level addressing structure. The software system exists in a segmented environment in which a logical address specifies the location of a data item. The processor automatically translates this logical address into a physical address for accessing the value in physical memory.

The memory occupied by an iAPX 432 software system is partitioned into many segments. Each segment is a group of contiguously addressed memory bytes that constitutes the physical representation of an object. Operands are always referenced in the data parts of objects. These data parts can have a maximum length of 65,536 bytes.

The instructions that make up the operations of a software system have access to the information contained within the objects that make up the current context access environment. Instructions may contain zero to three logical addresses each of which specifies the location of an operand in the data part of a directly accessible object. Operands are explicitly referenced by logical addresses that are encoded as Data Reference fields in instructions. Each data reference has two components: an access selection component and an operand offset component.

Figure 7-2 is a simplified overview of operand addressing.

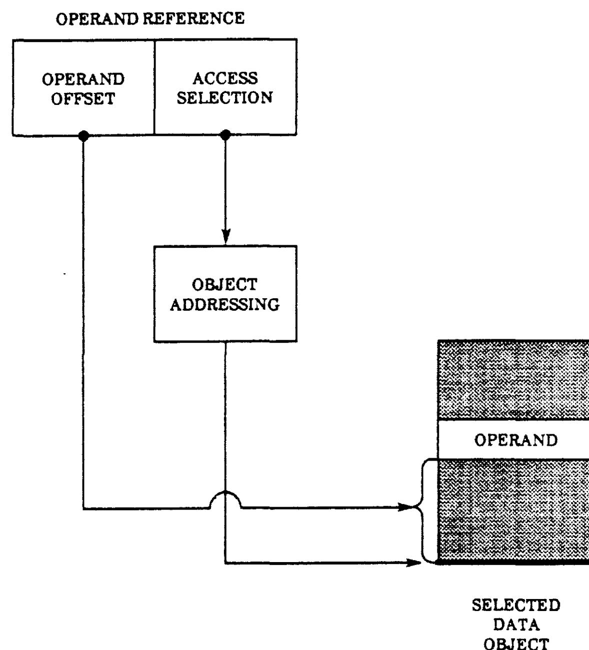


Figure 7-2. Operand Addressing Overview

The following two sections briefly describe the access selection and operand offset components. Later sections of this chapter describe in more detail the sub-components of these fields and discuss the semantics of their content. Object Addressing is given particular attention with regards to access rights and object type checking in the Object Addressing chapter of this manual. It is discussed in this chapter as it relates simply to operand addressing and physical address generation.

Access Selection Component

The access selection component specifies an index for an AD entry in one of the entered environment objects of the current context. The indexed access descriptor, in turn, references the object that contains the operand. The access selection component of a logical address can be specified directly in the data reference or indirectly via a value in the data part of an object. The value of a direct component must be known at compile time. An indirect component permits the value to be calculated dynamically at run time.

Operand Offset Component

The operand offset component specifies the offset into the data part of the selected object to the beginning byte of the desired operand. The byte offset is relative to the base or fence of the selected object. The offset is always a displacement into the data part of the selected object because operands are not interpreted in the access parts of objects. The operand offset component can be specified in more than one way. In addition to being specified directly in the data reference, it may also be determined indirectly by combining information in the data reference with values in the data parts of objects.

OPERAND ADDRESSING MODES

All operands reside in memory. There are no GDP registers visible to the programmer. Since a data reference is an encoding of an explicit logical address, it must provide both the access selection component and the operand offset component of the logical address. Both of these components can be specified in different ways. This flexibility provides a powerful addressing mechanism that allows efficient access to a variety of data structures. The addressing modes are completely orthogonal with respect to any of an instruction's operands. Any addressing mode is independently available to specify any required operand. This applies to all operators in the operator set.

The operand offset and access selection components independently contribute to the operand reference mode for a given operand. The modes for each of the two major components are described in detail in the following sections of this chapter. The data reference modes of the operand offset component are presented first. They are called data reference modes because each so closely relates to a particular kind of data structure in which the operand is located. After the components that make up the data reference modes are fully described, the access selection modes are presented.

Data Reference Modes

The encoding of the operand offset component of a data reference consists of two basic parts: a base part and an index part. This partitioning leads to viewing the entire data reference as having three components: an access selection component, which selects an object; a base part of the operand offset, which provides a byte displacement to the base of an area of memory within the selected object; and an index part of the operand offset, which specifies a particular operand within that area. This is illustrated in Figure 7-3.

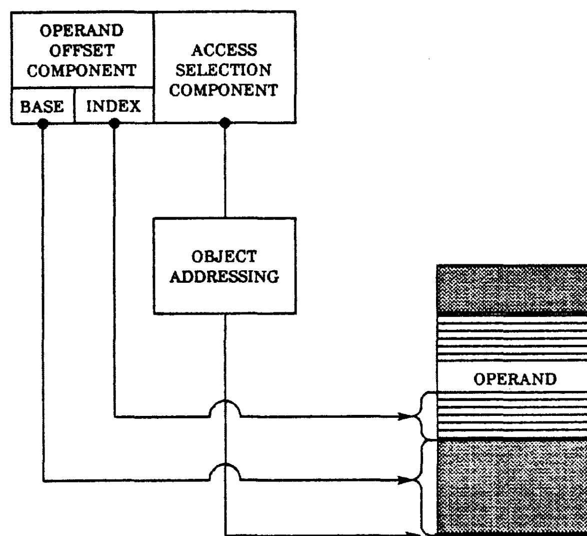


Figure 7-3. Base and Index Address Components

Flexibility is provided by allowing each part of the operand offset to be specified directly or indirectly. A direct base or direct index has its value specified directly in the data reference encoding. However, when indirection is used, the value of the base or index is given by a short-ordinal value located within a currently accessible object.

There are four possible combinations of direct and indirect base and index parts, and each combination results in a different mode of data reference. Figure 7-4 defines these four combinations. Figure 7-5 illustrates these four basic data reference modes, and subsequent sections describe each mode in detail.

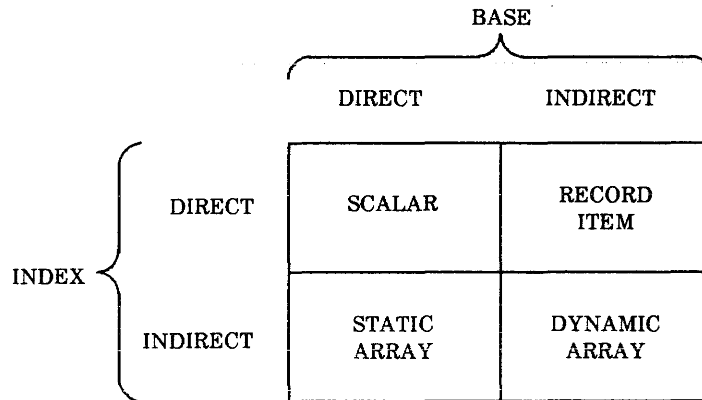


Figure 7-4. Data Reference Modes

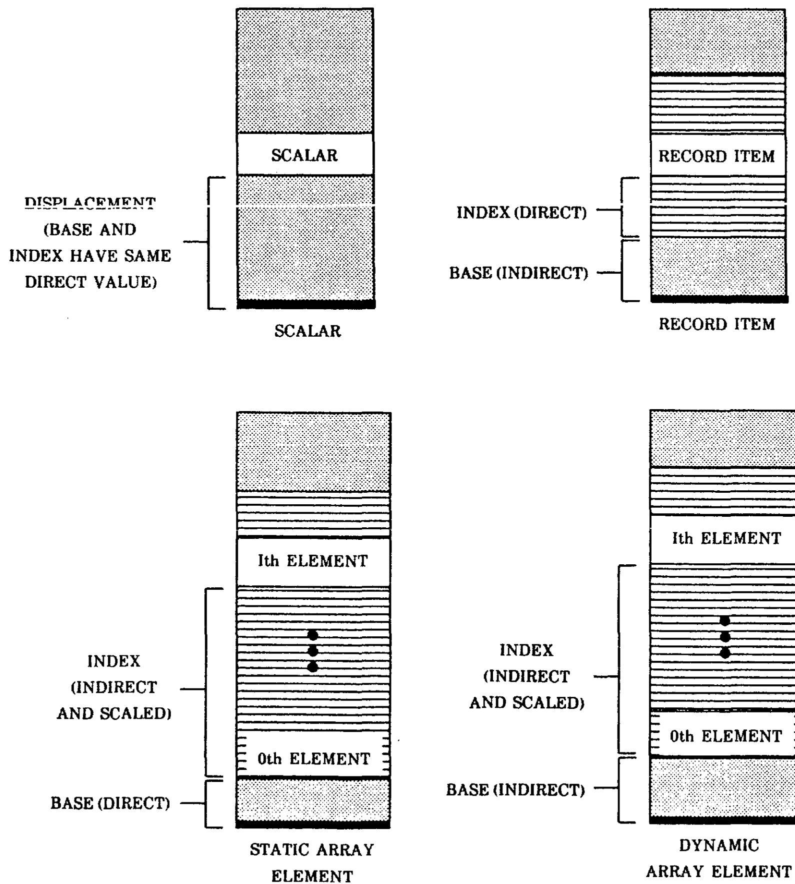


Figure 7-5. Data Reference Modes

Each of these four combinations has been used to name a data reference mode because each gives an indication of the kind of data structure for which the reference would typically be used (independent of the access selection mode). All four modes are independently available for any operand specified in an instruction. For each data reference mode, the processor calculates the operand offset as follows. (Bracketed items are specified indirectly.)

Operand Offset	=	displacement	(Scalar)
	=	[base] + index	(Record Item)
	=	base + [index]*scale	(Static Array Element)
	=	[base] + [index]*scale	(Dynamic Array Element)

Scalar Data Reference

A scalar data reference is the simplest type of data reference. It is used for direct access to operands of all the primitive computational data types. Figure 7-6 illustrates the scalar data reference mode.

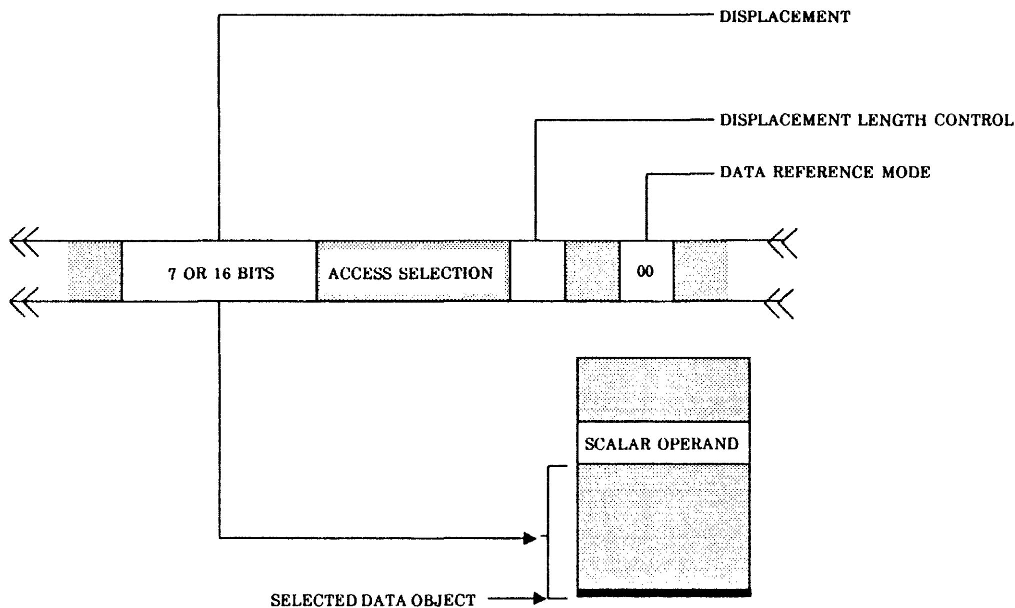


Figure 7-6. Scalar Data Reference

The Data Reference Mode field has an encoding of 00 for the scalar data reference mode. This field is the first two bits in every data reference. The Displacement Length Control field encodes one of two optional lengths for the Displacement field itself--either 7 or 16 bits long. The Displacement field encodes the byte displacement from the base (fence) of the selected object into the data part to the first byte of the scalar operand.
The 7-bit wide Displacement field is for operand offsets of less than 128 bytes. The 16-bit wide Displacement field is for operand offsets anywhere in the data part of the object. The Access Selection components encode information necessary to select the object in which the operand is located. Access Selection Modes are described in a later section of this chapter.

The GDP converts a scalar data reference to a logical address by first using the access selection component to select an object. The operand offset component is thus simply the value encoded in the Displacement field of the data reference.

Record Item Data Reference

Accessing a data item within an instance of a record requires three pieces of information: the data object in which the record instance is located, the byte displacement from the base of the object to the base of the particular record instance, and the byte displacement (Index) within the record to the data item. The Record Item Data Reference mode, illustrated in Figure 7-7, is designed for this kind of access.

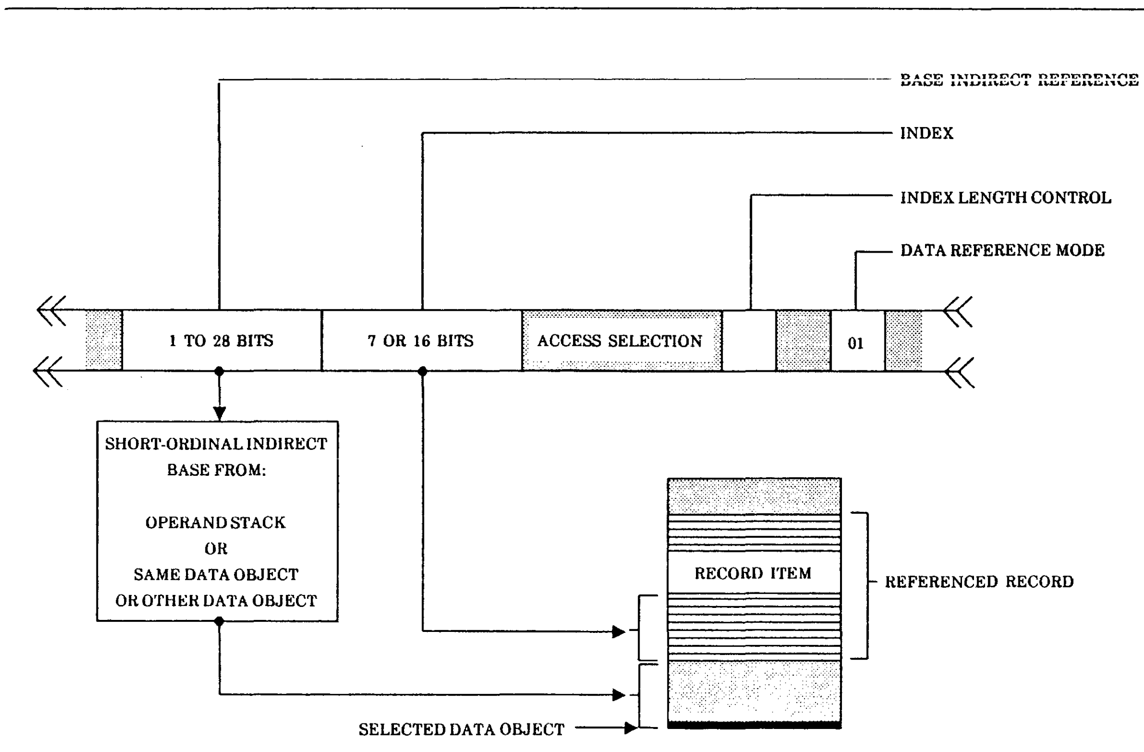


Figure 7-7. Record Item Data Reference

The access selection component for the entire data reference specifies the object in which the record instance is located. Access Selection is described in detail later in this chapter. The Base Indirect Reference field is encoded to specify the 16-bit indirect base from either the operand stack top, the same data object as that in which the referenced record is located, or some other data object. Indirect reference formats are described in detail later in this chapter.
The Index field is encoded directly in the data reference and is 7 or 16 bits long as determined by the Index Length Control bit. The Base Indirect Reference field encodes the byte displacement to the base of the record instance. This displacement is given indirectly so that the particular record instance to be accessed can be computed dynamically at run-time.

The GDP converts a record item data reference to a logical address by first using the access selection component to select an object. The operand offset component is then the sum of the values of the indirect base part and the direct index part. The addition operation uses 16-bit modulo arithmetic.

Static Array Element Data Reference

Three pieces of information are required to access an element of a static array: the object in which the array is located, the byte displacement from the base of the object to the base of the array, and the index of the particular array element. These three pieces of information are encoded in a Static Array Element Data Reference as shown in Figure 7-8.

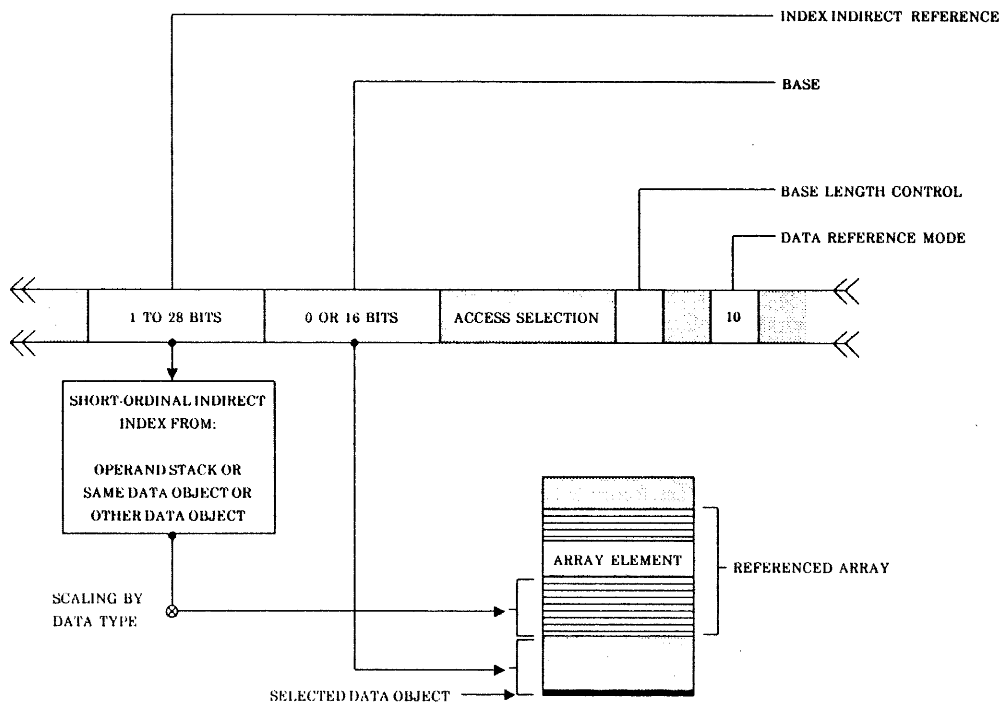


Figure 7-8. Static Array Data Reference

The Access Selection component (discussed later in this chapter) specifies the object in which the array is located. The byte displacement to the base of the array is encoded directly in the 0-bit or 16-bit Base field of the data reference. The 0- or 16-bit length of the Base field is determined by the Base Length Control bit. A 0-bit Base field means that the base part is not present in the data reference and that the value for the base is zero. The index to the array element (a byte displacement from the base of the array) is specified indirectly so that it can be computed dynamically at run-time.

If an index has the value i, it specifies the ith element from the base of the array, where an element can be any of the supported computational data types. The conversion of a static array data reference to an operand offset requires that this index value be converted to a byte displacement from the base of the array. The GDP automatically scales the index value, multiplying it by 1, 2, 4, 8, or 16 depending on whether the operand type occupies a byte, double-byte, word, double-word, or extended-word, respectively. Note that because of the manner in which scaling is done, when arrays of temporary-real operands are accessed with data references that automatically scale the index, each element is treated as if it were 16 bytes long. However, only the first 10 bytes of an element are actually read or written.

The GDP converts a static array element data reference to a logical address by using the access selection component to select an object. The operand offset component is then the sum of the values of the direct base part and the scaled value of the indirect index part. The addition operation uses 16-bit modulo arithmetic.

Dynamic Array Element Data Reference

Accessing an element of a dynamic array is the same as accessing an element of a static array except that the byte displacement to the base of the array may also be specified at run-time. This is the case if the array is located inside an object that is passed as a parameter to a procedure. This data reference mode is also useful in multiple dimension arrays where the base specifies a slice (row or column) of the multiple-dimension array. A data reference with both the base part and the index part specified indirectly is provided as shown in Figure 7-9.

The GDP converts a dynamic array element data reference to a logical address by using the access selection component to select an object. The operand offset component is then the sum of the values of the indirect base part and the scaled value of the indirect index part. The addition operation uses 16-bit modulo arithmetic.

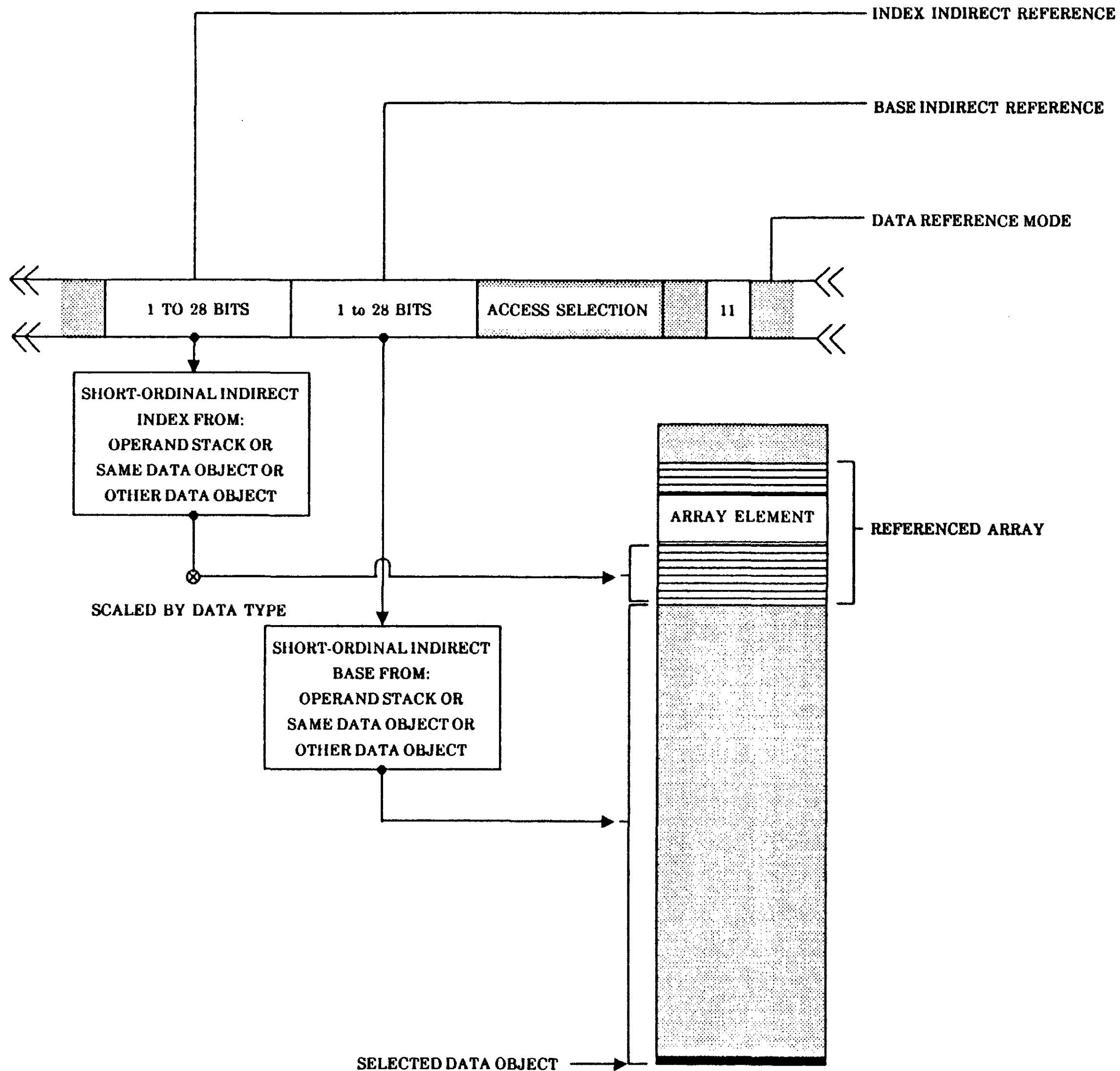


Figure 7-9. Dynamic Array Data Reference

Indirect Base and Index References

When indirection is used to specify the base or index parts of the operand offset component of a logical operand address, the data value that supplies the actual base or index value may be located in any one of three different ways: Stack Indirect Reference, Intrasegment Indirect Reference, and General Indirect Reference. All indirect values are 16-bit wide short ordinals.

Stack Indirect Reference. A stack indirect reference pops the base or index value on top of the operand stack. In this case, no additional encoding information is needed to locate the value, as is shown in Figure 7-10.

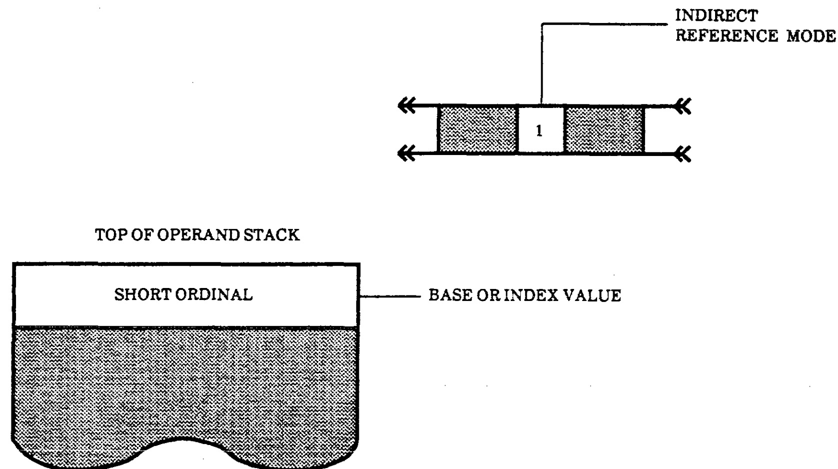


Figure 7-10. Stack Indirect Reference

Intrasegment Indirect Reference. An intrasegment indirect reference locates the base or index within the same object that is selected by the access selection component for the entire data reference. In this case, only the byte displacement from the base of the selected object to the base or index value needs to be encoded in the indirect reference. Figure 7-11 illustrates Intrasegment Indirect Reference.

General Indirect Reference. A general indirect reference locates the base or index value within any object accessible within the current context access environment. In this case, the indirect reference field contains a direct access selector and a byte displacement from the base of the selected object to the base or index value. This is the equivalent of a scalar data reference used to yield the short-ordinal base or index value. This is illustrated in Figure 7-12.

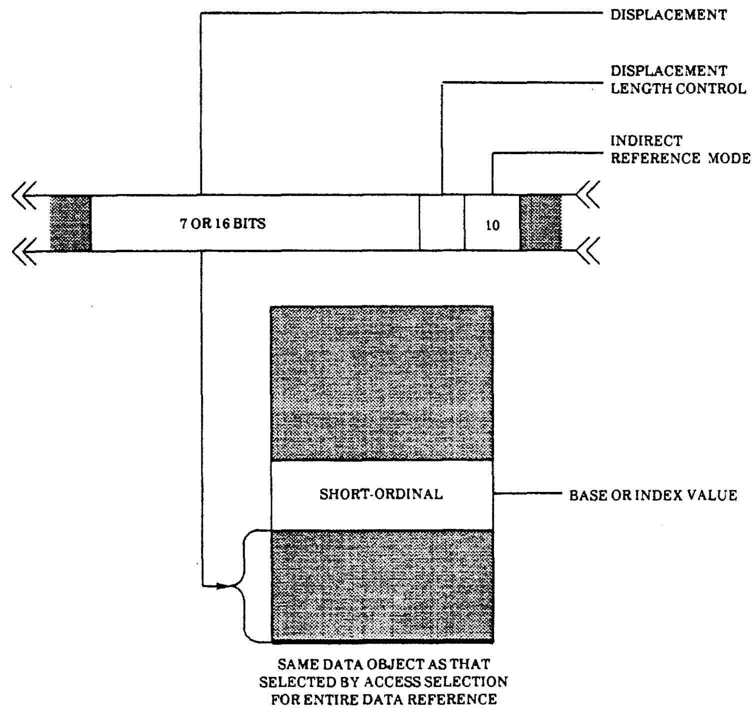


Figure 7-11. Intrasegment Indirect Reference

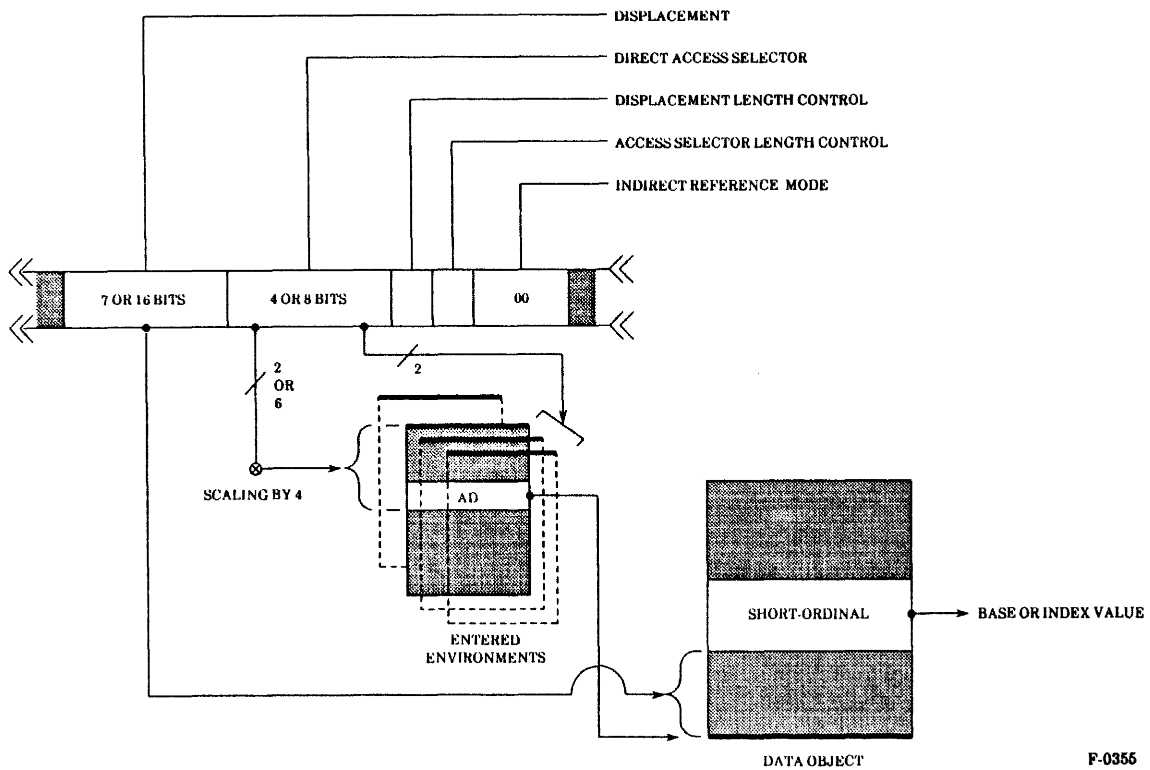


Figure 7-12. General Indirect Reference

Access Selection Modes

Each of the four basic data reference modes requires an Access Selection Mode field and an Access Selection field. The Access Selection Mode field specifies how the subsequent Access Selection field is to be interpreted. For the logical address encoded by a data reference, the access selection component can be specified in two different ways: by specifying a direct access selector or by specifying an indirect access selector. Figure 7-13 illustrates the access selection modes.

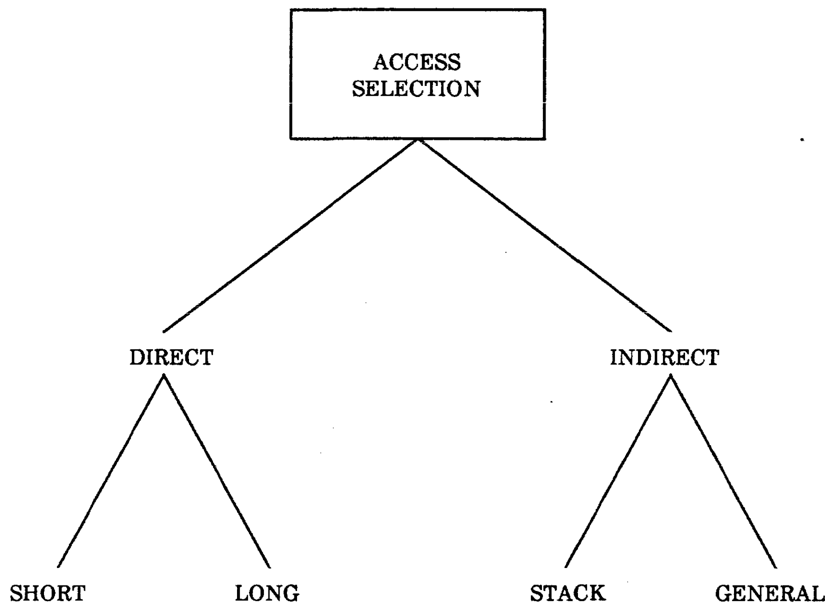


Figure 7-13. Access Selection Modes

Direct Access Selection

A direct access selector is encoded directly within the data reference as a 4- or 8-bit field. When the field is 4 bits, the mode is called Short Direct Access Selection; when the field is 8 bits, the mode is called Long Direct Access Selection. The Direct Access Selection Mode is illustrated in Figure 7-14.

The 0 encoded in the least-significant bit of the Access Selection Mode field specifies that the Access Selection field contains a direct access selector. The next-higher bit (in the Access Selection Mode field) is then used to differentiate the length of the direct access selector (0 for 4 bits, 1 for 8 bits).

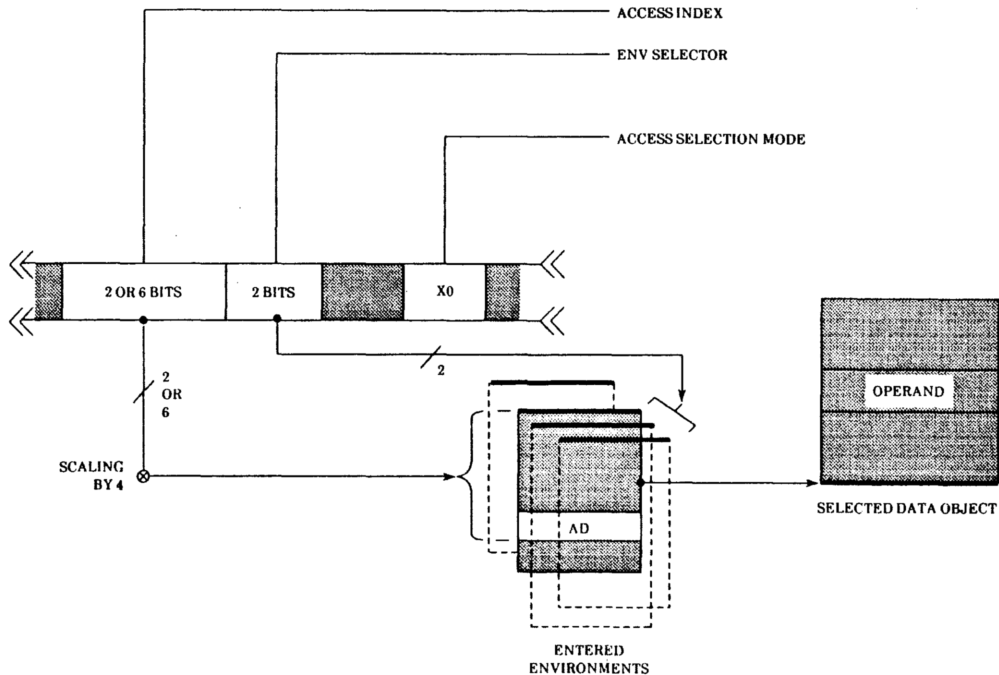


Figure 7-14. Direct Access Selection

As is evident in Figure 7-14, the low-order two bits (in the ENV Selector field) are interpreted to select one of the four Access Environment objects in the current context access environment. The remaining 2- or 6-bit access index provides an index into the access part of the selected object to an access descriptor. The AD in turn references the actual object in which the operand resides. For all access selection modes, the GDP automatically scales an access index by 4 (each AD is 4 bytes wide) to obtain the byte displacement into the access part to the selected AD.

The Short Direct Access Selector can specify the first four ADs in each environment. This covers the context, domain, global constants, and the context message--the majority of data references. The Long Direct Access Selector can specify the first 64 ADs in each environment. Since each environment may contain up to 16,384 ADs, AD entries beyond the first 64 can only be referenced by using the Stack Indirect Access Selection or the General Indirect Access Selection modes (described in the following sections).

If the access selector is specified indirectly, then the data reference contains information to locate a double byte within a currently accessible object. The value of this double byte is used as a 16-bit indirect access selector.

Stack Indirect Access Selection

This mode pops the access selector value from the current top of the operand stack. This mode only requires the Access Selection Mode field. The Access Selection field is omitted. This mode is shown in Figure 7-15.

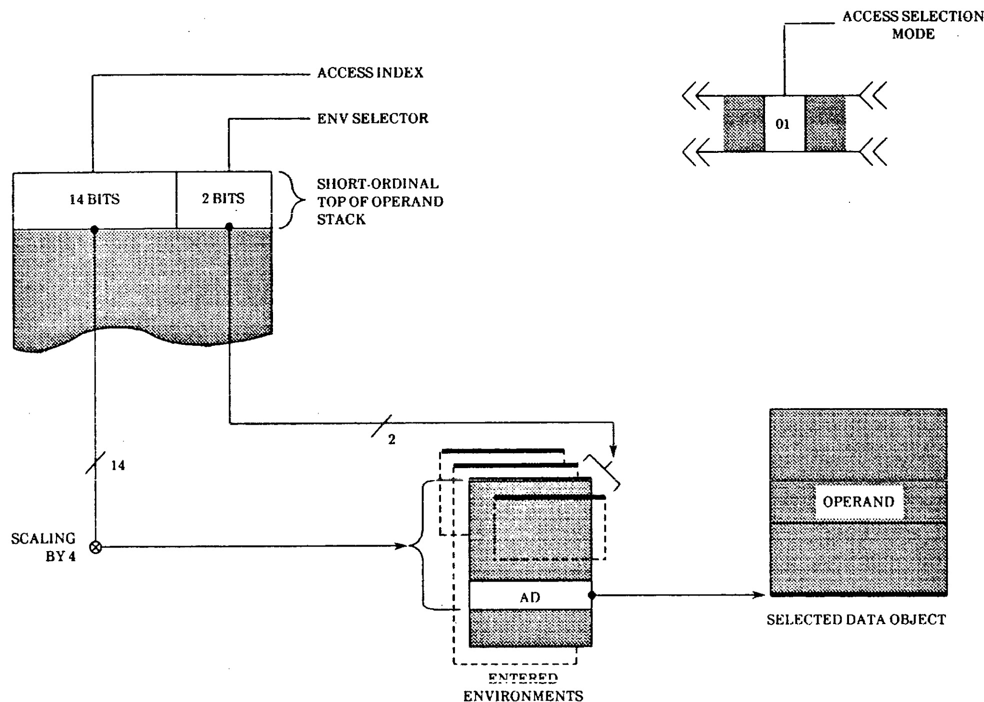


Figure 7-15. Stack Indirect Access Selection

In the same manner as described for direct access selectors, the low-order two bits (in the ENV Selector field) are interpreted to select one of the four Environments in the current context access environment. The remaining high-order 14 bits select an AD for the actual object in which the operand resides.

General Indirect Access Selection

The double byte that contains the indirect access selector can also be located in some other accessible object (i.e., other than the current context data part in which the operand stack is located). In this case, the Access Selection field contains both a direct access selector and a byte displacement. The direct access selector selects an object containing the double byte. The byte displacement accesses that double byte within the selected object. The accessed double byte then contains the indirect access selector that selects the object in which the operand resides. This mode of access selection is called General Indirect Access Selection Mode and is illustrated in Figure 7-16.

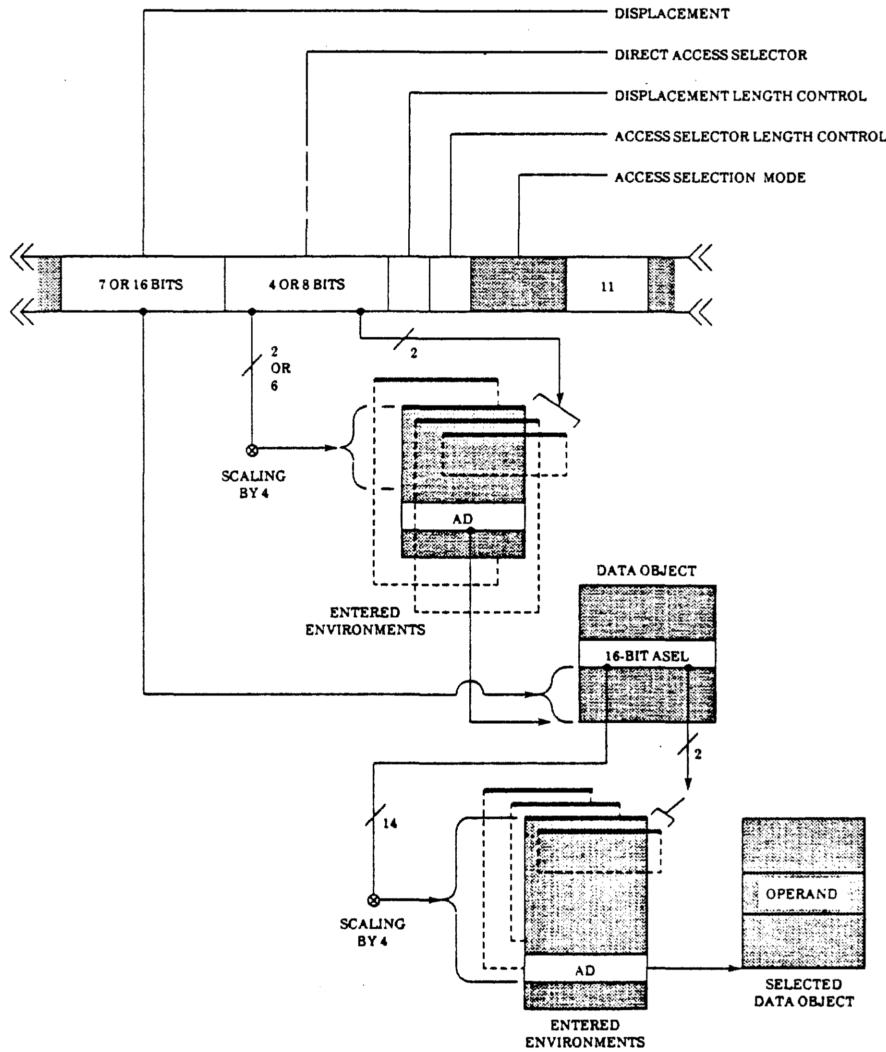


Figure 7-16. General Indirect Access Selection

BRANCH REFERENCES

The GDP operator set includes several branch operators. Some require operands to indirectly specify the target instruction of the branch. These operands are referenced as described earlier in this chapter. Note that a branch reference is not an operand. It is a directly encoded part of a branch instruction (in the Reference field) that determines the bit address of the instruction that is the target of the branch. There are two types of branch references: relative branch references and absolute branch references.

A relative branch reference is encoded by a 10-bit signed integer value. It is used as the bit displacement to the target instruction relative to the beginning of the branch instruction. Thus, the range of a relative branch is from -512 to +511 bits relative to the first bit of the branch instruction itself. No wraparound occurs. This means that if the displacement overflows or underflows the boundary of the current instruction object, a Type 1 Instruction Pointer Overflow fault occurs.

An absolute branch reference is encoded by a 16-bit short ordinal value that is the bit displacement from the base of the current instruction object to the target instruction.

The ranges of relative and absolute branches are illustrated in the Figure 7-17.

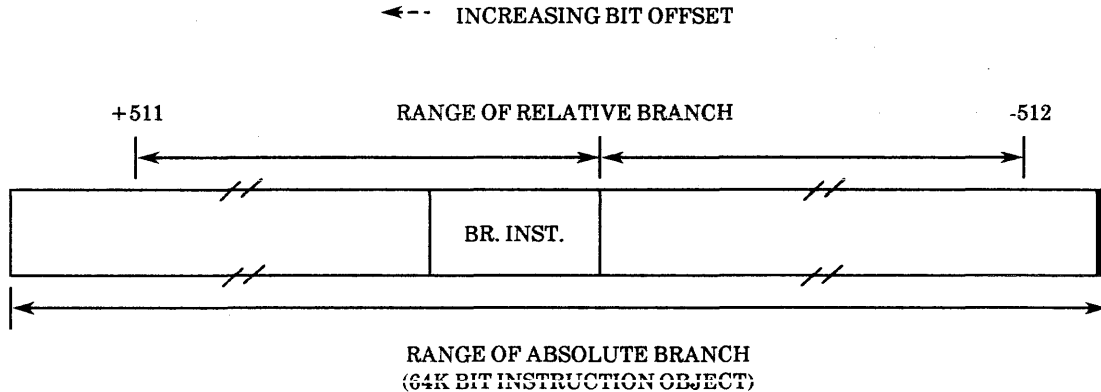


Figure 7-17. Branch References

LARGE ARRAY INDEXING

The maximum size of the data part of an object is 65,536 bytes (64K). Some applications require arrays larger than 65,536 bytes. The GDP operator set provides the Index Ordinal operator to support accessing such large arrays.

The large array is mapped (at creation time) into a series of objects, each with data parts that are 2,048 bytes (2K) long. These objects are directly accessible in the current logical access environment. The Index Ordinal operator works as follows:

Given:

- The size of each element in the array (i.e., a scale factor)
- The access selector for the base segment of the array
- The ordinal index for the desired array element

The operator computes:

- The access selector for the appropriate 2K data object that contains the indexed array element
- The displacement into the data part of that object to the array element

These resultant short-ordinal values can then be used with the indirect access selection mode and the record item, static array, or dynamic array data reference modes to access the array element. The range for index values must always begin at 0. This means that array bounds such as -2000 to +500 must be accomplished by explicitly biasing the index. Thus, the compiler must map the source level array bounds onto an ordinal range starting at 0. This ordinal is then used by the compiler as the indirect source index operand of a compiled Index Ordinal instruction.

Note that only the least-significant four bits of the scale factor are used. The array element size is two raised to the power of this 4-bit value. The permissible array element sizes are thus: 1, 2, 4, 8, 16, 32, 64, ... 32,768 bytes. Accordingly, if the application's array element size is 20 bytes, the scale factor should be 5 (element size of 32). If the application uses only the least-significant 20 bytes of each 32-byte element, the most-significant 12 bytes are not used.

OPERAND STACK INTERACTION

Embedded in the data part of each context object is the context's dedicated operand stack. Any operand of an instruction may be specified as the value (of appropriate data type) at the top of the operand stack.

The operand stack is handled in a uniform manner whenever implicit stack references are used in an instruction. Unless the operator is one of the SAVE operators, using the operand stack to obtain a source operand causes the value to be popped from the stack when it is fetched. The SAVE operators provide the ability to read the value at the top of the operand stack without popping it. The SAVE operator will also duplicate the top stack value on the stack when the stack is specified as both the source and destination. There are a number of indivisible operators which push the original value of the destination on top of the operand stack as an inherent part of their operation.

Whenever the operand stack is specified as the destination, the result of the operation is pushed onto the stack. If the operand stack is used as the source for both source operands of an order-three operator, the Format field in the instruction specifies the order in which the two operands appear on the stack. Source operands are popped from the top of the operand stack, and destination operands are pushed onto the operand stack.

The stack pointer, cached in a register within the GDP, contains the displacement in bytes from the base of the current context object (into the data part) to the first free byte on the operand stack. When an operand is pushed onto the stack, the value is stored beginning at the location specified by the stack pointer, and the stack pointer is then incremented by the length of the operand. Similarly, when an operand is popped from the stack, the stack pointer is first decremented by the length of the operand, and then the value is read beginning at the location specified by the stack pointer. The operand stack is 16 bits wide; the stack pointer is maintained on double-byte boundaries.

The GDP caches the access information for the current context object. The operand stack pointer is maintained by an internal register to specify the displacement to the current top. The top double-byte of the operand stack is also cached. An instruction referencing the top operand on the stack need specify neither an access selector nor an operand offset. Since very little information is required in an instruction to encode a stack reference, temporary results of operations are most efficiently stored on the operand stack.

The exact manner in which operands are stored in an operand stack depends on the length of the operand. A byte operand stored on the stack is stored in the low byte of a double-byte stack element. A double-byte operand simply occupies one double-byte wide element on the stack. Word, double-word, and extended-word operands require two, four, and five stack elements, respectively, with higher-order elements stored at higher addresses.

During the interpretation of a given instruction, the operand stack may be the source for several data values. Some of these values may be used in forming logical addresses, while others may be used as actual source operands. Because of this multiple use of the stack, the order in which operand addresses are formed and values are popped from the stack is specified in the following paragraphs. In general, items are popped (during instruction decoding) in the order in which they are encountered in the instruction.

There can be no operand order ambiguity with order-zero operators. This is also true of most order-one operators, with the exception of the SAVE operators. SAVE operators read a value from the operand stack without popping the stack, and store that value in the destination operand specified by the single data reference. As with all instructions containing order-one operators, the single data reference is evaluated before the SAVE operation is actually executed. As a result, if the data reference has an indirect access selector, an indirect base part, or an indirect index part that is located in the operand stack, that addressing information must be on top of the stack with the value to be SAVEd immediately below. When the SAVE operation is actually performed, the addressing information will have already been used and popped from the stack, leaving the value to be saved as the top stack element.

If a single data reference has more than one indirect part located on the operand stack, the following ordering conventions are used. If there are three indirect parts on the operand stack, the base part is assumed nearest the top, the index part is immediately below the base part, and the indirect access selector is immediately below the index part. If only two indirect parts are on the operand stack, this same ordering applies with the missing part deleted.

In the case of order-two operators, the address of data reference one is formed first. Since it is always a source operand, the associated value is fetched, and then the address of operand two, normally a destination address, is formed. To understand the importance of this ordering, consider, for example, an instruction that moves a value from the stack to a location specified by a static array element data reference whose index part is also on the stack. Since the value of the first operand is fetched before the address of the second operand is formed, the operand stack must have the value to be moved in the top stack element and the index for the destination in the next-to-top stack element.

For order-three operators, the address of operand one is formed first, and the associated value is fetched. Then the address of operand two is formed, and the associated value is fetched. Finally, the address of operand three--normally a destination--is formed, and any result is stored there.

INSTRUCTION INTERPRETATION

Before the individual instructions in an instruction object can be interpreted by a GDP, appropriate process, domain, context, and instruction object addressing information must be loaded into the processor registers. The appropriate system objects are "qualified" by taking physical access information from each corresponding object descriptor and loading it into the appropriate internal registers. These registers are not visible to the programmer. Among other things, their contents form an access environment in which logical addresses from the instruction stream can be translated to the physical addresses that reference memory. Once a processor dispatches itself to execute a process, and a proper access environment has been established, the processor begins to interpret the instruction referenced by the current value of the instruction pointer.

The instructions for a GDP are encoded into a common format that allows the interpretation of each instruction to proceed in a single, logical sequence. An instruction consists of a series of fields (described earlier in this chapter) that are organized to present information to the processor as it is needed. As an instruction is fetched from memory, each of its several fields is decoded. Execution of the specified operation proceeds basically as follows:

- Decode the number of operands and the length of each (using the Class field)
- Decode the operand addresses, convert them to physical addresses, and fetch the source operands (using the Format and Reference fields)
- Decode the operator, perform it, and store any result (using the Reference and Opcode fields)

Because the instruction stream of the GDP is a bit stream, instructions are a variable number of bits in length and are not aligned on byte or word boundaries. In the current GDP implementation, instruction fetches are 32-bit memory accesses issued whenever the decoder needs more bits. These 32-bit fetches are made independent of the alignment of the instructions and independent of the current instruction execution. Instruction decoding and execution are asynchronous and pipelined; while the current instruction is executing, the next section of the instruction stream can be undergoing fetch and decode by the processor.

PHYSICAL ADDRESS GENERATION

A 432 software system exists in an object-based environment in which a logical address specifies the location of an operand within an object. The processor automatically translates this logical address into a physical address for accessing the operand in physical memory. User programs have no way to generate physical addresses directly. This section describes the processor activity of physical address generation.

Note that to accelerate this address generation, the current implementation of the architecture uses two address caches to store a set of the most recently used addresses. This avoids a pass through some or all of the memory-resident descriptor structures for each access to a qualified (address cached) object.

Figure 7-18 illustrates physical address generation and is followed by a general algorithmic description.

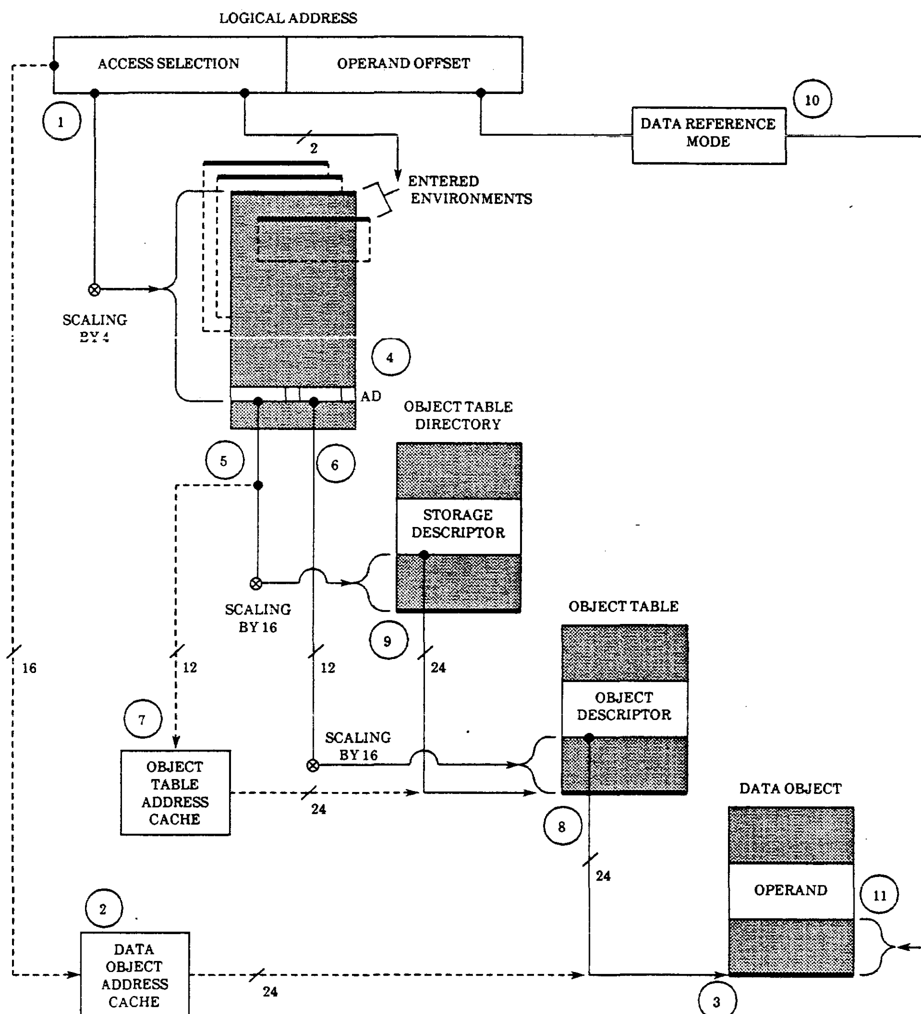


Figure 7-18. Physical Address Generation

The following algorithmic description uses numbered notes that are keyed to the preceeding illustration. This algorithmic description is not meant to be exhaustive. For example, type and rights checking are not included here. They are discussed in the Object Addressing chapter of this manual. The procedure begins with a logical address (as specified in a data reference) and yields the 24-bit physical address of the first byte of the referenced operand.

- The specified Access Selector (1) is used to search the Data Object Cache (2).
    - If a match is found (indicating that the Data Object (3) has been previously cached), the corresponding 24-bit physical address from the Data Object Cache entry is used to locate the Data Object in which the operand resides.
    - If no match is found, the access selector is used in the normal way to locate an AD in the current access environment (4). Note that the specified access selection mode of the current data reference can entail a recursion of Physical Address Generation for each instance of an access selector in the access selection mode.
        - The Directory Index (5) in this AD is used to search the Object Table Cache (7).
            - If a match is found (indicating that the indexed Object Table (8) has been previously cached), the 24-bit physical address from the Object Table Cache entry is used to locate the indexed Object Table.
            - If no match is found, the Object Table is located normally through the Object Table Directory (9) by using the Directory Index (5). When this is the case, the 24-bit physical address for the Object Table is loaded (with other information) into an appropriate entry in the Object Table Cache.
        - The Segment Index (6) from the AD (4) is used to index into the Object Table (8) to the OD for the selected Data Object (3). This OD is then used to provide a 24-bit physical base address for the Data Object. When this is the case, the 24-bit physical base address is loaded (with other information) into an appropriate entry in the Object Cache (2).
- The operand offset is calculated using the specified data reference mode. This calculation itself can entail a recursion of Physical Address Generation for each instance of an access selector in the data reference mode.
- The calculated operand offset (10) is added to the physical base address of the Data Object (3) to obtain the final physical address of the first byte of the operand (11).

The physical address is then transmitted to memory by the processor. A physical address is always 24 bits wide. This results in a maximum physical memory size of 16 Megabytes (16,777,216 bytes).

INSTRUCTION EXECUTION

Under normal conditions, a GDP is controlled by instructions that are fetched in sequence from the current instruction object on behalf of the current process being executed. An instruction is fetched from the current instruction object by using a bit displacement obtained from the instruction pointer that is maintained within the GDP. After the instruction has been executed, the instruction pointer is incremented by the number of bits in the instruction so that it points to the next sequential instruction.

Under abnormal conditions, any action in response to software and hardware exceptions takes several forms with the GDP. Most computational faults cause a simple branch within the faulting context to a designated fault handler instruction object. More severe faults, such as object protection faults, result in suspending the execution of the faulting process and in sending an access descriptor for its process object to a fault service process via a designated fault port. A faulting processor is handled somewhat differently by reassigning it to a different dispatching port where it may attempt to execute a programmed diagnostic process. For more information on fault handling see the Fault and Trace Management chapter of this manual.

The GDP has two different kinds of operators. Data operators operate on zero, one, or two operands of computational data types and produce a single result--also a computational data type--which is stored as the destination operand. Object operators perform high-level manipulation of system objects, typically incurring a transformation on the memory image of one or more objects. Data operators do not reference or alter information other than the primitive data items specified by the source and destination operands. However, object operators may access and alter information in a number of objects during execution.

CHAPTER 8
COMPUTATIONAL DATA TYPES

This chapter describes the computational data types that are recognized by the GDP. The data types are first introduced by describing their sizes and uses. Then follows a section discussing the GDP's operators for manipulating the data types. Next, the GDP's floating-point model is presented in detail, along with an explanation of floating-point operand interpretation. Finally, Data Operator faults are classified and described.

OVERVIEW OF COMPUTATIONAL DATA TYPES

Figure 8-1 summarizes the sizes and general uses of the GDP's computational data types.

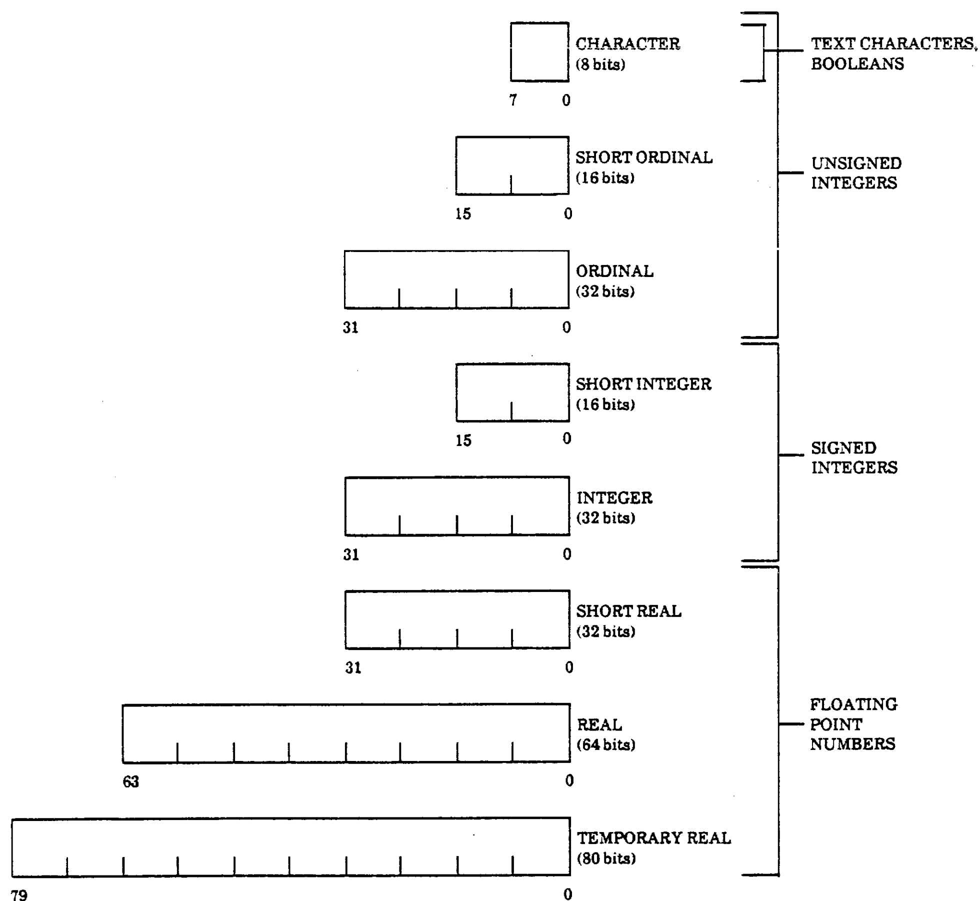


Figure 8-1. Computational Data Types

The following sections briefly introduce the computational data types and describe the numerical ranges that the types can represent.

CHARACTER DATA TYPE

This 8-bit data type is used to represent Booleans, text characters, or unsigned integers.

A Boolean is a value used to represent logical TRUE or FALSE. TRUE is represented by xxxxxxx1 and FALSE is represented by xxxxxxx0 (with the x bits being uninterpreted "don't-care" bits).

Text characters are used to make up text data. Arrays of text characters can be used for software-defined string data types.

For unsigned integers, the character data type is used to represent values in the range 0 to 255.

SHORT-ORDINAL DATA TYPE

This 16-bit data type represents unsigned integer values in the range 0 to 65,535, or bit fields of 1 to 16 bits.

ORDINAL DATA TYPE

This 32-bit data type represents unsigned integer values in the range 0 to 4,294,967,295, or bit fields of 1 to 32 bits.

SHORT-INTEGER DATA TYPE

This 16-bit data type represents signed integer values in the range -32,768 to 32,767 in two's complement form.

INTEGER DATA TYPE

This 32-bit data type represents signed integer values in the range -2,147,483,648 to 2,147,483,647 in two's complement form.

SHORT-REAL DATA TYPE

This 32-bit data type represents floating point numbers. Normalized short-real values provide the equivalent of aproximately 7 decimal digits of precision. Their interpretation as operands by the GDP is described in detail later in this chapter.

REAL DATA TYPE

This 64-bit data type represents floating point numbers. Normalized real values provide the equivalent of aproximately 15 decimal digits of precision. Their interpretation as operands by the GDP is described in detail later in this chapter.

TEMPORARY REAL DATA TYPE

This 80-bit data type represents floating point numbers. Normalized temporary-real values provide the equivalent of approximately 19 decimal digits of precision. Their interpretation as operands by the GDP is described in detail later in this chapter.

OPERATORS FOR COMPUTATIONAL DATA TYPES

This section presents an overview of the GDP operators provided for the computational data types. The Operator Set chapter describes these operators in detail. Figure 8-2 illustrates the computational operators and which data types they apply to.

DATA TYPES

	OPERATORS	CHARACTER	SHORT ORDINAL	ORDINAL	SHORT INTEGER	INTEGER	SHORT REAL	REAL	TEMPORARY REAL
MOVE OPERATORS	MOVE	X	X	X	X	X	X	X	X
	SAVE	X	X	X	X	X	X	X	X
	ZERO	X	X	X	X	X	X	X	X
	ONE	X	X	X	X	X	-	-	-
LOGICAL OPERATORS	AND	X	X	X	-	-	-	-	-
	INCLUSIVE OR	X	X	X	-	-	-	-	-
	EXCLUSIVE OR	X	X	X	-	-	-	-	-
	EQUIVALENCE	X	X	X	-	-	-	-	-
	NOT	X	X	X	-	-	-	-	-
ARITHMETIC OPERATORS	ADD	X	X	X	X	X	*	*	X
	SUBTRACT	X	X	X	X	X	*	*	X
	MULTIPLY		X	X	X	X	*	*	X
	DIVIDE		X	X	X	X	*	*	X
	REMAINDER		X	X	X	X	-	-	X
	INCREMENT	X	X	X	X	X	-	-	-
	DECREMENT	X	X	X	X	X	-	-	-
	NEGATE	-	-	-	X	X	X	X	X
	ABSOLUTE VALUE	-	-	-			X	X	X
	SQUARE ROOT								X
	INDEX	-	-	X	-		-	-	-
BIT-FIELD OPERATORS	EXTRACT		X	X	-	-	-	-	-
	INSERT		X	X	-	-	-	-	-
	SIGNIFICANT BIT		X	X	-	-	-	-	-
RELATIONAL OPERATORS	EQUAL	X	X	X	X	X	X	X	X
	NOT EQUAL	X	X	X	X	X			
	EQUAL ZERO	X	X	X	X	X	X	X	X
	NOT EQUAL ZERO	X	X	X	X	X			
	LESS THAN	X	X	X	X	X	X	X	X
	LESS THAN OR EQUAL	X	X	X	X	X	X	X	X
	POSITIVE	-	-	-	X	X	X	X	X
	NEGATIVE	-	-	-	X	X	X	X	X
	MOVE IN RANGE				X	X			
CONVERSION OPERATORS	TO CHARACTER	-				X			
	TO SHORT ORDINAL	X	-			X			
	TO ORDINAL			-		X			X
	TO SHORT INTEGER				-	X			
	TO INTEGER	X	X	X	X	-			X
	TO SHORT REAL						-		X
	TO REAL							-	X
	TO TEMPORARY REAL		X	X	X	X	X	X	-

WHERE:
- X MEANS THE OPERATOR IS AVAILABLE FOR THE GIVEN DATA TYPE.
- * MEANS THE OPERATOR IS AVAILABLE FOR THE GIVEN DATA TYPE AND FOR INSTRUCTIONS IN WHICH ONE OF THE OPERANDS IS A TEMPORARY REAL.
- \- MEANS THE OPERATOR IS NOT AVAILABLE AND WOULD BE OF LITTLE OR NO USE IF IT WERE.
- (BLANK) MEANS THE OPERATOR IS NOT AVAILABLE.

F-0273

Figure 8-2. Operators and Data Types

BIT FIELD MANIPULATION

A special set of operators is provided to manipulate bit fields in short-ordinal and ordinal operands. The extraction operators (EXT_SO, EXT_O) allow a specified bit field in a short-ordinal or ordinal source operand to be extracted and right justified (with high-order zeros) to form a corresponding short ordinal or ordinal result. The insertion operators (INS_SO, INS_O) allow a specified field in a short-ordinal or ordinal destination to be written with the right-justified binary value of a corresponding short-ordinal or ordinal source operand. The significant bit operators (SIG_SO, SIG_O) allow the most significant set bit of a short-ordinal or ordinal source operand to be determined and stored as a position value in a short-ordinal destination operand. Figure 8-3 illustrates bit field insertion and extraction operations.

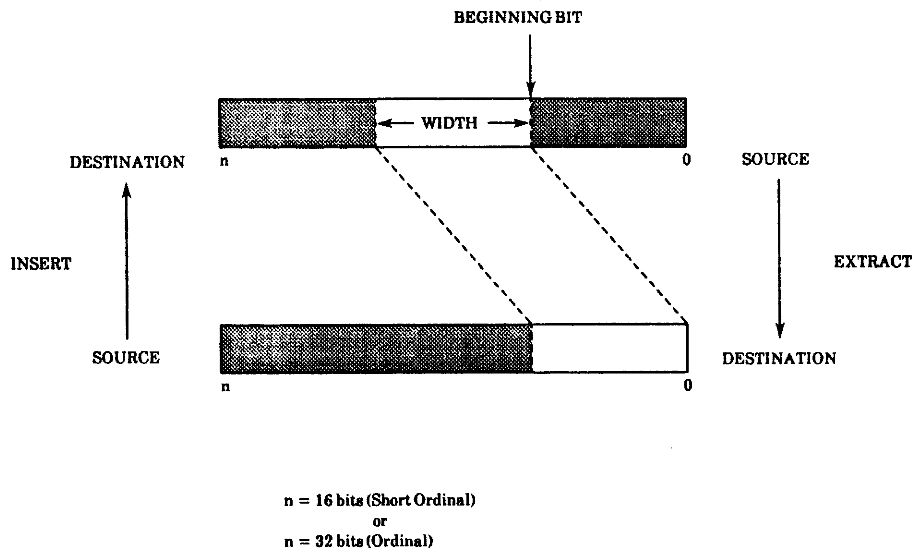
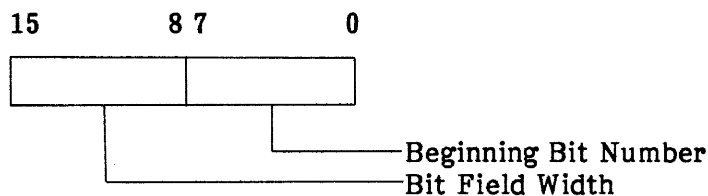


Figure 8-3. Bit-Field Operations

The insertion and extraction operators use a special type of operand, a <u>bit field specifier</u>, to specify the extracted source or inserted destination field of the operation. A bit-field specifier consists of two adjacent bytes as shown:



For short-ordinal operators, only the low-order 4 bits of these bytes are interpreted by the GDP during execution. For ordinal operators, only the low-order 5 bits of these bytes are interpreted by the GDP during execution. The two bytes are interpreted as follows:

Beginning Bit Number (bits 0 - 7)
: The first byte specifies the beginning bit of the field. The bits of an operand are numbered with bit number 0 being the least-significant bit. The beginning bit of a field is the smallest numbered bit in the field.

Bit Field Width (bits 8 - 15)
: The second (next higher-addressed) byte specifies one less than the number of bits in the field. For short-ordinal operators, a field of any width up to 16 bits can be specified by a bit-field specifier, regardless of the beginning position. For ordinal operators, a field of any width up to 32 bits can be specified by a bit-field specifier, regardless of the beginning position. If a field is specified that extends beyond the most-significant bit of the operand, bit 0 is considered to follow the most-significant bit in a wrap-around fashion.

DATA TYPE CONVERSION

The GDP's operator set includes several operators to allow conversion between the various computational data types, as shown in Figure 8-4.

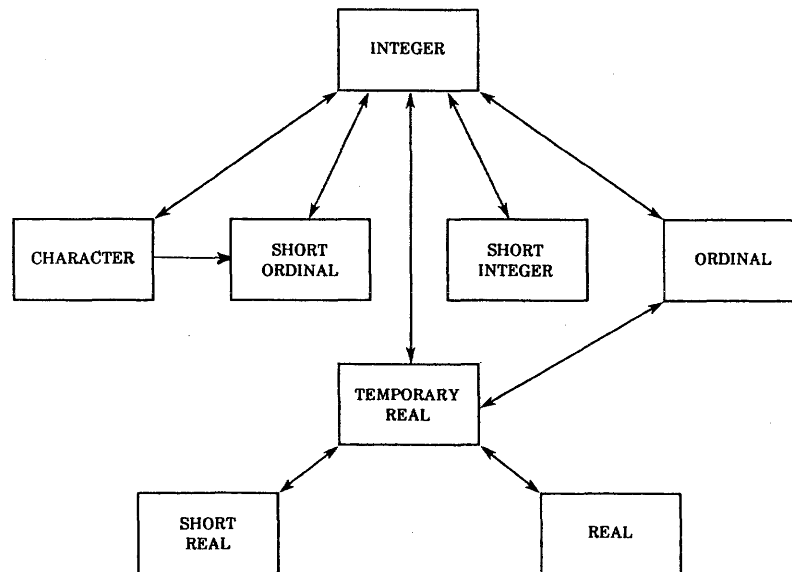


Figure 8-4. Data Type Conversions

GDP FLOATING-POINT DATA TYPES

The GDP directly supports the major time-critical aspects of the IEEE Proposed Standard for Binary Floating-Point Arithmetic. In most cases, a single machine instruction performs a given floating-point operation to completion in accordance with the IEEE standard. However, certain requirements of the standard must be provided by software enhancements to the underlying floating-point architecture of the GDP. In particular, the GDP faults on denormalized, NaN (Not a Number), infinity, and pseudo-denormalized operands. In certain cases, unnormalized temporary reals will also cause a fault. By raising these faults, the GDP can invoke an appropriate fault handler that provides the necessary computations to complete the faulted operation in accordance with the IEEE standard. The proposed IEEE standard is presented in:

> Floating-Point Working Group, Microprocessor Standards Committee, IEEE Computer Society, "A Proposed Standard for Binary Floating-Point Arithmetic", Draft 8.0 of IEEE Task P754. Computer, March 1981, pp. 51-62.

Both the IEEE proposed standard single precision (32-bit) and double precision (64-bit) floating point formats are supported by the GDP. In addition, a double-extended (80-bit) temporary format is provided.

The GDP thus recognizes three floating-point data types:

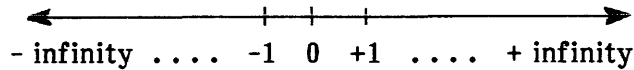
- Short-Real, occupying 4 bytes each (corresponding to the IEEE standard single-precision format)
- Real, occupying 8 bytes each (corresponding to the IEEE standard double-precision format)
- Temporary-Real, occupying 10 bytes each (corresponding to the minimal IEEE standard double-extended format)

Each data type is characterized by the amount of storage required and the operators available for operands of that type. The set of operators provided permits the accurate determination of almost any arithmetic operation to true 64-bit precision.

The GDP supports floating-point computations that involve the manipulation of short-real, real, and temporary-real operands by the set of operators associated with these data types. The floating-point architecture has been designed to provide clean, accurate arithmetic for floating-point computations and to ease the writing of reliable mathematical software.

In the discussion that follows, the term "floating-point" is used generically to describe short-real, real, and temporary-real data types and their associated operators. Several special terms are also used (e.g., denormalized, unnormalized, infinity, and NaN). They are defined only briefly later in this chapter. See the IEEE proposed standard for a more complete discussion of these terms. See also the iAPX 86/20, 88/20 Numerics Supplement in the iAPX 86, 88 User's Manual.

GDP calculations using floating-point operands are essentially approximations to ideal calculations carried out on values from the set of real numbers. From a mathematical viewpoint, the set of real numbers can be viewed as a number line stretching from minus infinity to plus infinity:



- infinity .... -1 0 +1 .... + infinity

Each point from the infinite set of points on this line represents a unique real number. The real numbers that can be represented exactly by short-real, real, and temporary-real operands form a finite set of discrete points along this number line. Note that the values represented by character, short-ordinal, ordinal, short-integer, or integer operands also form a finite set of discrete points along this number line. All of the values that can be represented exactly by these data types can be represented exactly by real or temporary-real operands, and most can be represented exactly by short-real operands.

The design of the GDP floating-point architecture is based on a particular model of floating-point computation. This model assumes that any computation begins with a set of source values that are represented by either short-real or real operands. The programmer selects one of the representations for the values, depending on the precision or exponent range required. The values produced by the computation are also represented by short-real or real operands. Again, the programmer makes the choice, depending on the precision or exponent range required in the results. The model further assumes that any intermediate result that is generated during the computation is represented by a temporary-real operand. The additional precision provided by the temporary-real values allows for much more precision in the final results than if short-real or real values were used as intermediates. The extended exponent range of temporary reals allows computations to continue that might otherwise have been halted because the exponent range of a short-real or real operand was not sufficient to hold the exponent of an intermediate result.

The set of GDP operators associated with the floating-point data types is distributed to support the computational model described above. All of the order-three arithmetic operators produce results that are temporary-real values. Also, each of these arithmetic operators has three forms, which allows mixing the precisions of the source operands. For example, there are three operators for addition of short-real operands. One adds two short-real operands, another adds a short-real operand to a temporary-real operand, and the third adds a temporary-real operand to a short-real operand. All three produce a temporary-real result as their third operand.

The advantage of this type of operator distribution can be seen from the following Ada program fragment that might be used as part of a statistical calculation. It calculates the sum of the elements of a vector and the sum of the squares of the elements:

```
SUM := 0.0
SUMSQ := 0.0
for I in 1 .. MAXI loop
     SUM  := SUM + A(I);
     SUMSQ  :=  SUMSQ + A(I) * A(I);
end loop;
```

Assume that variable A represents an array of short-real values, that I is a short-ordinal index, and thast MAXI is the maximum index for A. The variable SUM accumulates the array sum, and the variable SUMSQ accumulates the sum of the squares.

The ease with which the GDP can support this Ada fragment is illustrated by a translation of the fragment into the following hypothetical assembly language sequence:

```
            ZRO_TR      SUM                 ; SUM := 0.0;
            ZRO_TR      SUMSQ               ; SUMSQ := 0.0;
            ONE_SO      I                   ; Start I at 1
<<LOOP>>    ADD_TR_SR   SUM,A(I,SUM         ; SUM := SUM + A(I);
            MUL_SR      A(I),A(I),$         ; Push A(I) * A(I)
            ADD_TR      $, SUMSQ,SUMSQ;     ; SUMSQ := SUMSQ+<STK>
            INC_SO      I,I                 ; Increment index I
            LEQ_SO      I,IMAX,$            ; I <= MAXI ?
            BR_T        <<LOOP>>            ; yes, then loop
```

The order of the operands shown in the above instructions is the order defined for the appropriate operators in this manual. The \$ indicates that the operand is pushed onto, or popped from, the operand stack.

The ADD_TR_SR, MUL_SR, and ADD_TR operators are order three operators that product temporary-real results as their third operand. Thus, the two results (SUM and SUMSQ) are best represented as temporary-real values to maintain as much precision as possible during the calculations in the loop.

In accordance with the IEEE standard, the GPD floating-point architecture also provides the programmer with complete control over rounding. The SET CONTEXT MODE operator allows setting the bits in the current context status. Rounding Control is used to determine the number of significant digits to which the rounding is performed. The SET PROCESS MODE operator allows setting the Inexact Control bit in the process status. Inexact control provides additional flexibility by allowing the programmer to select faulting on an inexact result.

Conversion operators are also provided to allow conversion between all of the GDP's floating-point formats using two conversion instructions at most. On most cases, only a single conversion instruction is required.

GENERAL FLOATING-POINT FORMAT

As a short-real, real, or temporary-real operand, the floating-point representation of a number consists of three binary fields:

These fields are interpreted by the GDP as follows.

S
: This one-bit field represents the number's algebraic sign.

Exponent
: This field represents the number's binary order of magnitude.

Significand
: This field represents the number's significant digits. The Significand is often broken down into its Most Significant Bit (MSB), an implicit binary radix point, and a Fraction field.

CLASSIFICATION OF FLOATING-POINT NUMBERS

The following terms serve to classify various kinds of floating-point numbers recognized by the GDP. For a more detailed discussion of these terms, see the IEEE proposed standard cited at the start of this chapter.

Normal — a short-real, real, or temporary-real number with a nonzero, nonmaximum exponent and a significand MSB of 1.

Zero — (normal zero) a short-real, real, or temporary-real number with a zero exponent and zero significand.

Denormal — a short-real, real, or temporary-real number with a zero exponent, a significant MSB of 0, and a nonzero fraction.

NaN — (Not a Number) a short-real, real, or temporary-real number with a maximum exponent and a nonzero fraction.

Infinity — a short-real, real, or temporary-real number with a maximum exponent and a zero fraction.

Unnormal — a temporary-real number with a nonzero, nonmaximum component, an explicit significand MSB of 0, and a nonzero fraction.

Pseudo-zero a temporary-real number with a nonzero, nonmaximum exponent, and a zero significand.

Pseudo-denormala temporary-real number with a zero exponent and an explicit significand MSB of 1.

NORMALIZED FLOATING-POINT NUMBERS

The GDP performs floating-point arithmetic using normalized floating-point numbers. In most cases, if short-real or real operands are not normalized, the GDP interprets them as invalid operands and raises the Domain Error Fault. With a few exceptions, the GDP interprets unnormalized temporary real values as valid operands.

A normalized number has a significand with a most significant bit if 1 and a nonzero, nonmaximum exponent. Normalization allows the maximum number of significant digits to be represented by a significand of a given width, because leading zeros are eliminated. This maximizes the precision accommodated by the represented floating-point number; it ensures that high-order zeros in the fraction are shifted out and compensated for by decrementing the exponent by 1 for each shift left. Thus, the binary number:

+ 0.00000 00000 00000 00001 11111

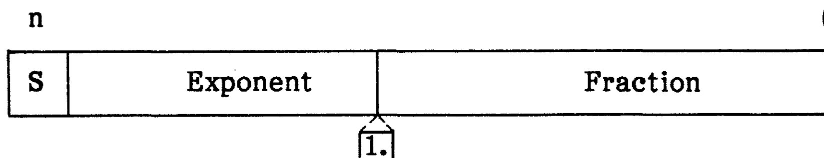
can be equivalently represented as:

+ 1.11111 * (2 ** -20)

where, in both representations, "." is the binary radix point, which is the binary counterpart of a decimal point. The -20 value would constitute the true value that must be represented in the binary Exponent field.

The most significant bit of the significand, called the leading bit, is implicit in the normalized short-real (32-bit) and real (64-bit) data types. That is, when a short-real or real operand is referenced, the GDP assumes a leading "1." at the high-order (leftmost) end of the significand field. For short-real and real operands, this implicit leading bit is only interpreted as 0 when the Exponent field is 0.

A normalized floating-point number with the implicit leading bit and binary point is shown below.



The Fraction field is defined as that portion of the significand immediately to the right of the binary point.

The number of significant bits in a short-real number or real significand is therefore one greater than the bit-field width of the physically stored fraction.

Temporary-real formats do not use an implicit leading bit in the significand; the leading bit (with the implicit binary point following it) is explicit and physically present. A normalized temporary-real operand must have an explicit, leading significand bit of 1.

Table 8-1 summarizes the significand sizes for the GDP's floating-point data types.

Table 8-1 Significand Sizes

DATA TYPE	No. of Bits in Explicit Fraction	Is There an Implicit Bit Interpreted?	Total Number of Bits Contributing to Significand
Short Real	23	Yes	64
Real	52	Yes	53
Temporary Real	64	Explicit	64

EXPONENT BIASES

In accordance with the IEEE standard and to obtain closure under multiplicative inverse (i.e., 1/x neither overflows nor underflows), the GDP interprets floating-point exponents as being biased by a constant value. Table 8-2 summarizes the sizes and biases for the Exponent fields for the GDP's three floating-point data types.

Table 8-2 Exponent Sizes and Biases

DATA TYPE	No. of Bits in Exponent	Maximum in Field	Minimum in Field	Bias of Exponent
Short Real	8	255	0	127
Real	11	2047	0	1023
Temporary Real	15	32767	0	16383

Biases are constant values that are automatically added to the true exponent to force the biased exponent to always be a positive value. A number's true exponent can thus be determined by subtracting the bias value for that type from the stored exponent value:

True Exponent = Biased Exponent - Bias

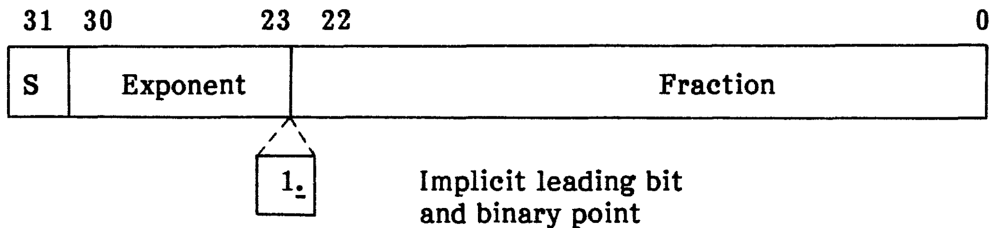
When exponents are biased, two normalized floating-point representations of the same type and sign can be compared as if they were simple binary magnitudes. That is, when comparing them bitwise beginning with the most significant exponent bit, the first position that differs serves to order the numbers; no further comparison need occur. This ease of comparison is one of the benefits of biasing the exponents.

GDP FLOATING-POINT OPERAND INTERPRETATION

The following sections describe the unique aspects of each floating-point data type when interpreted by the GDP as an operand.

Short-Real Operand Interpretation

Short-real operands occupy 32 bits of storage and have the following format:



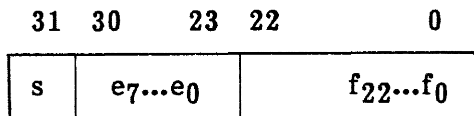
The most significant bit (bit 31) specifies the sign of the represented number (0 is positive, 1 is negative).

There are three classes of short-real operands: normalized operands, zero operands, and invalid operands.

Normalized short-real operands are those in which the Exponent field is neither all zeros nor all ones. The binary significand is stored in a true magnitude form that assumes an implicit bit with value 1 to the left of the most significant bit of the Fraction field. The implicit binary point is between this implied leading bit and the most significant fraction bit.

Values in the Exponent field have an unsigned binary integer range from 0 through 255. The unsigned binary integer in the Exponent field is interpreted as though a bias of 127 had been subtracted from it. Since an Exponent field of all zeros or all ones (unsigned integer values of 0 or 255) has a special meaning, the true exponent range represented is -126 through 127.

The value of the normalized short-real operand stored as



is calculated by

$$(-1)^s * (1.f_{22}...f_0) * (2^{(e_7...e_0)-127})$$

Normalized short-real operands provide the equivalent of approximately 7 decimal digits of precision. The smallest and largest decimal absolute values are approximately:

smallest: $1.2 * 10^{-38}$
largest: $3.4 * 10^{38}$

A zero short-real operand has both a zero exponent and a zero Fraction field. The implicit leading bit is assumed to be zero also. Signed short-real zeros are interpreted. The interpretation of signed zero floating-point operands is discussed later in this chapter.

Invalid short-real operands always cause a Type 0 Domain Error fault (Invalid Operand) when referenced in an arithmetic, relational, or conversion instruction. Though the GDP does not interpret invalid operands beyond recognizing them and raising the Domain Error fault, Table 8-3 suggests a classification that can be used by an appropriate fault handler to fulfill the requirements of the IEEE standard. The shaded areas are invalid operands.

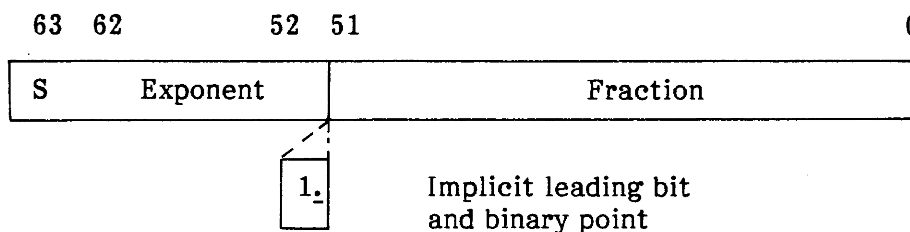
Table 8-3 Short-Real Operand Classifications

SIGNIFICAND	EXPONENT		
	Zero	From 1 to Max-1	Max
Zero	±Zero	Normalized	±Infinity
Nonzero	Denormalized	Normalized	NaN

For further information on infinity arithmetic, denormal arithmetic, and NaNs, see the IEEE proposed standard that is referenced at the start of this chapter. See also the iAPX 86/20,88/20 Numerics Supplement in the iAPX 86,88 User's Manual.

Real Operand Interpretation

Real operands occupy 64 bits of storage and have the following format:



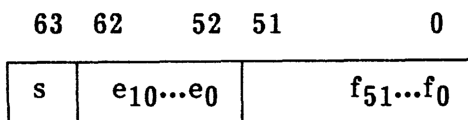
The most significant bit (bit 63) specifies the sign of the represented number (0 is positive, 1 is negative).

As with short-reals, there are three classes of real operands: normalized operands, zero operands, and invalid operands.

Normalized real operands are those in which the Exponent field is neither all zeros nor all ones. The binary significand is stored in a true magnitude form that assumes an implicit bit with value 1 to the left of the most significant bit of the Fraction field. The implicit binary point is between this implied leading bit and the most significant fraction bit.

Values in the Exponent field have an unsigned binary integer range from 0 through 2047. The unsigned binary integer in the Exponent field is interpreted as having a bias of 1023 subtracted from it. Since an Exponent field of all zeros or all ones (unsigned integer values of 0 or 2047) has a special meaning, the true exponent range represented is -1022 through 1023.

The value of the normalized real operand stored as



is calculated by

$$(-1)^s * (1.f_{51}...f_0) * (2^{(e_{10}...e_0)-1023})$$

Normalized real operands provide the equivalent of approximately 15 decimal digits of precision. The smallest and largest decimal absolute values are approximately:

smallest:	$2.2 * 10^{-308}$
largest:	$1.8 * 10^{308}$

A zero real operand has both a zero exponent and a zero Fraction field. The implicit leading bit is assumed to be zero also. Signed real zeros are interpreted. The interpretation of signed zero floating-point operands is discussed later in this chapter.

Invalid real operands always cause a Type 0 Domain Error fault (Invalid Operand) when referenced in an arithmetic, relational, or conversion instruction. Though the GDP does not interpret invalid operands beyond recognizing them and raising the Domain Error fault, Table 8-4 suggests a classification that can be used by an appropriate fault handler to fulfill the requirements of the IEEE standard. The shaded areas are invalid operands.

Table 8-4 Real Operand Classifications

SIGNIFICAND	EXPONENT		
	Zero	From 1 to Max-1	Max
Zero	±Zero	Normalized	±Infinity
Nonzero	Denormalized	Normalized	NaN

Temporary-Real Operand Interpretation

Temporary-real operands occupy 80 bits of strage and have the following format:

The most significant bit (bit 79) specifies the sign of the represented number (0 is positive, 1 is negative).

Temporary-real operands are intended for use as intermediate, or temporary results during floating-point computations. Temporary-reals correspond to the minimal double-extended format in the proposed IEEE standard. Supporting such temporary results has two very important benefits:

- The use of temporary-real operands for the intermediate values of a multi-step calculation allows a result to be obtained with much less loss of precision than would occur if short-real or real operands were used to hold the intermediate values.

- The extended exponent range greatly reduces the possibility that overflow or underflow might occur and halt the computation before it is complete.

As with short-reals and reals, there are three classes of temporary-real operands: normalized operands, zero operands, and invalid operands.

Normalized temporary-real operands are those in which the Exponent field is neither all zeros nor all ones. Unlike the interpretation of short-real and real operands, the interpretation of normalized temporary-real operands does not involve an implicit leading bit in the significand. Instead, the binary significand is stored in a true magnitude form that assumes an explicit most significant bit with a value of 1. The implicit binary point is interpreted as being to the immediate right of the explicit leading one bit.

Values in the Exponent field have an unsigned binary integer range from 0 through 32767. The unsigned binary integer in the Exponent field is interpreted as though a bias of 32767 had been subtracted from it. Since an Exponent field of all zeros or all ones (unsigned integer values of 0 or 32767) has a special meaning, the true exponent range represented is -16382 through 16383.

The value of the normalized temporary-real operand is stored as

79	78 ... 64	63 ... 0
s	$e_{14} \ldots e_0$	$f_{62} \ldots f_0$

is calculated by

$$(-1)^s * (1.f_{62} \ldots f_0) * (2^{(e_{14} \ldots e_0)-1023})$$

Normalized real operands provide the equivalent of approximately 19 decimal digits of precision. The smallest and largest decimal absolute values are approximately:

smallest: $1.7 * 10^{-4932}$
largest: $1.2 * 10^{4932}$

A zero temporary-real operand has both a zero exponent and a zero Significand field. The implicit leading bit is assumed to be zero also. Signed temporary-real zeros are interpreted. The interpretation of signed zero floating-point operands is discussed later in this chapter.

Temporary-real invalid operands always cause a Type 0 Domain Error fault (Invalid Operand) when referenced in an arithmetic, relational, or conversion instruction. Though the GDP does not interpret invalid operands beyond recognizing them and raising the Domain Error fault, Table 8-5 suggests a classification that can be used by an appropriate fault handler to fulfill the requirements of the IEEE standard. The shaded areas are invalid operands.

Table 8-5 Temporary Real Operand Classifications

SIGNIFICAND		EXPONENT		
MSB	Fraction	Zero	From 1 to Max-1	Max
1	Nonzero	Pseudo-Denorm	Normalized	NaN
1	Zero	Pseudo-Denorm	Normalized	±Infinity
0	Nonzero	Denormalized	Unnormalized	NaN
0	Zero	±Zero	Pseudo-Zero	±Infinity

Unnormalized and Pseudo-Zero temporary reals are invalid only when used as source operands for the following operators (* means only invalid as denominator):

- Arithmetic: SQT_TR, DIV_TR*, REM_TR*, DIV_TR_SR*, DIV_TR_R*
- Relational: EQL_TR, EQZ_TR, LSS_TR, LEQ_TR, PTV_TR, NTV_TR
- Conversion: CVT_TR_O, CVT_TR_I, CVT_TR_SR, CVT_TR_R

True Remainder for Temporary Reals

Though a REMAINDER TEMPORARY REAL operator is available for temporary-real operands, it does not perform the complete remainder calculation. This section describes the remainder function in general and how to calculate the true remainder for two temporary real numbers using the available REM_TR operator.

The behavior of the remainder calculation is best described in terms of an example. Consider the problem of dividing 2102.5 by 51 using decimal arithmetic. The calculation, using long division, is:

```
                                         41.
                                   51/ 2102.5
                                       204
    first partial remainder  ------------>62.5
                                          51
    second partial remainder ------------>11.5
```

Each step of the division algorithm generates one digit in the quotient and a partial remainder. The remainder of interest is the partial remainder that results when the last digit of the integer part of the quotient has been generated (the second partial remainder in the example). Note that this partial remainder is the first one that is less than the divisor. Note also that as the value of the dividend increases, or the value of the divisor decreases, the number of digits that must be generated in the integer part of the quotient increases.

The calculation of the true remainder requires performing steps of the division algorithm until the last digit of the integer part of the quotient has been generated. In the case of temporary-real operands, the number of required division steps could be quite large--so large as to be impractical within one instruction execution. The REMAINDER TEMPORARY REAL operator provides the basic capability that allows the complete remainder function for temporary reals to be programmed in software. The time associated with executing this operator is approximately the same as that for a normal temporary-real division.

The result of the REMAINDER TEMPORARY REAL operator does not depend on the setting of the Rounding Control bits or the Precision Control bits in the Context Status field. The result or partial result generated is always exact; accuracy is not lost in the process. Each time the REMAINDER TEMPORARY REAL (REM_TR) operator is executed, one step of the division algorithm is performed. It can be iterated in a program loop until a fixed number of division steps have been performed or until a partial remainder is generated whose absolute value is less than the absolute value of the divisor. In the latter case, the partial remainder is the true remainder. The following program fragment illustrates the true remainder calculation loop in a hypothetical assembly language:

```
         MOV_TR   DIVIDEND,PARTREM           ; PARTREM := DIVIDEND
LOOP:    ABS_TR   PARTREM,$                  ; push ABS of PARTREM
         ABS_TR   DIVISOR,$                  ; push ABS of DIVISOR
         LSS_TR   $2,$1,$                    ; ABS(PARTREM) less than
         BR_T     $,DONE                     ; ABS(DIVISOR)?
                                             ; if yes, then exit
         REM_TR   DIVISOR,PARTREM,PARTREM    ; else calc next PARTREM
         BR       LOOP                       ; and do loop again
DONE:                                        ; PARTREM contains result.
```

The order of the operands shown in the above instructions is the order defined for the appropriate operators in the Operator Set chapter of this manual. $ indicates that the operand is pushed onto, or popped from, the operand stack. $1 and $2 indicate that the operand is popped from the top ($1) or next-to-top ($2) of the operand stack.

Zero as a Floating-Point Operand

Both positive and negative zero operands are interpreted by the GDP for all three floating-point data types, as follows:

- a sign bit of 0 for positive, 1 for negative
- an exponent field of all 0s
- a significand field of all 0s

Both positive and negative zero are distinguished by the GDP to allow more information to be associated with certain results than if only a single zero was recognized. For example, when performing interval arithmetic, the interval (+0,N) can indicate that the value zero is not included in the interval. Similarly, the interval (-0,N) can indicate that the value zero is included.

However, the notion of two distinguishable zeros has a more important use than for interval arithmetic. For example, the result of the operation A/0, where A is a positive value, can be considered the limit of A/x as x approaches 0. The actual limit depends on whether 0 is approached from the positive or negative side of the number line. If 0 is approached from the positive direction, the result becomes more and more positive as x approaches 0, and can be represented by positive infinity. If 0 is approached from the negative direction, the result becomes more and more negative as x approaches 0, and can be represented by negative infinity. These two different limit operations can be represented by A/+0 and A/-0.

In general, any operation that involves zero as an operand can be thought of as a limit operation, and the sign of the zero is positive or negative depending on whether the limit is to be taken by approaching zero from the positive or negative side of the number line, respectively. Similarly, if zero is the result of a limit operation, the sign indicates the direction from which zero was approached as the limit was taken. For example, let A be positive and consider dividing A by positive infinity. This represents the limit of A/x as x approaches positive infinity. The result can be represented by +0, since A/x approaches zero from the positive side of the number line. Likewise, A divided by negative infinity can be represented by -0.

All of the GDP's floating point operators provide correct results when a zero of either sign occurs as a source operand. Also, when a zero result is produced, the correct sign is produced. Table 8-6 shows the results for all arithmetic and relational floating-point operators that can have signed zero operands. All combinations of source operands that involve values of zero or produce results of zero are shown for these operators.

Table 8-6 does not show the floating-point ZERO (ZRO) operators, which always store +0. The table also does not show the conversion (CVT) operators. The following rules apply for floating-point zeros in conversions:

- Conversion from non-floating-point to floating-point:
  If the source operand is zero, the result is +0.

- Conversion from floating-point to floating-point:
  -0 is converted to -0, and +0 is converted to +0.

- Conversion from floating-point to non-floating-point:
  Both +0 and -0 are converted to non-floating-point zero.

As an example of using the Signed Zeros table, consider the result of subtracting -0 from -0. Looking in the SUB column at the row in which OP1 and OP2 are both -0, the entry +0* is found. It means (as indicated by the *) that the result depends on the rounding mode:

- -0 if the rounding mode is Round Down

- +0 for any other rounding mode

Table 8-6. Signed Zeros

OP1	OP2	ADD	SUB	MUL	DIV	REM	SQT	NEG	ABS	EQL	EQZ	LSS	LEQ	PTV	NTV
+0	+V	+V	+V	+0	--	--	+0	-0	+0	F	T	T	T	F	F
-0	+V	+V	+V	-0	--	--	-0	+0	+0	F	T	T	T	F	F
+0	-V	-V	-V	-0	--	--				F		F	F		
-0	-V	-V	-V	+0	--	--				F		F	F		
+V	+0	+V	-V	+0	+0	+0				F		F	F		
-V	+0	-V	+V	-0	-0	+0				F		T	T		
+V	-0	+V	-V	-0	-0	-0				F		F	F		
-V	-0	-V	+V	+0	+0	-0				F		T	T		
+0	+0	+0	±0*	+0	--	--				T		F	T		
+0	-0	±0*	-0	-0	--	--				T		F	T		
-0	+0	±0*	+0	-0	--	--				T		F	T		
-0	-0	-0	±0*	+0	--	--				T		F	T		
+V	-V	±0*													
+V	+V		±0*												

NOTES

1. V represents an arbitrary nonzero positive floating-point value.
2. T is the Boolean value TRUE; F is FALSE.
3. For entries marked ±0*, the result is -0 if the rounding mode is Round Down and +0 for all other rounding modes.
4. -- entries are invalid operations (division by zero).
5. For Order-2 operators, OP1 is the single source operand.
6. For Order-3 operators, OP1 and OP2 are the source operands and correspond to the same-named operands in the Operator Set chapter of this manual.

FLOATING-POINT ROUNDING

Rounding Modes

As described earlier in this chapter, floating-point operands represent discrete points along the real number line. When certain floating-point operators are executed, the true value of the result may be a value that cannot be represented by a floating-point operand of the type produced by the operator. In this case, the result must be rounded before it is stored in the result operand, and the result is said to be inexact. It must be rounded to one of the two representable floating-point values on either side of the true result. The GDP's floating-point architecture provides explicit control over the manner in which this rounding is done. The option is also provided to fault on the occurrence of an inexact result and thus to allow fault-handling software to provide a warning or to complete the operation by dealing with the inexact result in a user-programmed way.

Four rounding modes are supported: Round Nearest, Round Up, Round Down, and Round Toward Zero. Their selection is controlled by appropriate Rounding Control bits in the Context Status field of the current context object. The Context Status of the current context can be changed by a special SET CONTEXT MODE operator.

Round Nearest is the normal kind of rounding used. With Round Nearest, the true result is rounded to the nearest representable floating-point value. If the true result lies along the number line exactly halfway between the two representable values on either side of it, then the true result is rounded to the representable value with a least significant fraction bit of zero. This rounding mode delivers the most statistically unbiased results and is recommended for normal use.

The Round Up and Round Down modes make it easy to program interval arithmetic. Interval arithmetic requires that each step of a floating-point computation produce an interval (i.e., an upper and lower bound) that is certain to contain the true result. The two types of rounding required for interval arithmetic are thus round toward positive infinity (Round Up) and round toward negative infinity (Round Down). Whenever the true result of a floating-point operator lies between two representable floating-point values, rounding up will produce the algebraically larger value, and rounding down will produce the algebraically smaller value. If the true result falls exactly on a representable value, then both rounding up and rounding down produce the same true result.

Rounding Toward Zero (chopping) chooses the representable value that is algebraically closest to zero. If the true result falls exactly on a representable value, then rounding produces the same true result.

Rounding Control

The rounding control used in the current context can be changed with the SET CONTEXT MODE operator. The new rounding mode is local to the context (does not propogate to the caller) but is inherited by called contexts. Note that it is not proper to simply write a new value to the Context Status field that contains the control bits, as this field is cached by the GDP.

Precision Control

The Precision Control bits in the Context Status field control the precision to which temporary-real results are rounded. These bits determine whether the result fraction is to be rounded to temporary-real precision (64 bits), real precision (53 bits) or short-real precision (24 bits). As described in the previous section, the direction of rounding used is specified by the Rounding Control bits. If a temporary-real result is rounded to real precision, the low-order 11 fraction bits are zero; if rounded to short-real precision, the low-order 40 fraction bits are zero.

The Precision Control bits are assigned in the same way as the Rounding Control bits and with the same scope, by using the SET CONTEXT MODE operator. This operator assigns the control bits in both the executing GDP and in the Context Status field.

Inexact Control

The programmer can control whether or not a fault is invoked when an inexact result is generated. Inexact fault-handling can be used to allow user-programmed provisions for inexact results in certain numeric algorithms, or for the use of temporary reals to simulate a 64-bit long integer data type.

Unlike the rounding control and precision control bits, the Inexact Control bit is located in the Process Status field and is controlled by the SET PROCESS MODE operator.

DATA OPERATOR FAULTING

CLASSIFICATION OF DATA OPERATOR FAULTS

If an operation is attempted with an operand that is not defined for that operation, or if the true result of any operand violates the representable bounds of the destination operand, then an exceptional condition is recognized by the GDP and an appropriate context-level fault is raised. All Data Operator faults are Type 0 faults and are classified into the following fault groups:

- Domain Error -- caused by an exceptional operand value being outside the numeric domain that is defined for the attempted operation (e.g., square root of a negative number, attempt to divide by zero, floating-point invalid operand).

- Overflow -- caused by a true result with an absolute value that exceeds the maximum representable value of the actual destination operand (e.g., attempting DEC_SI on a source operand value of -32,768, attempting INC_SI on a source operand value of 32,767).

- Underflow -- caused by a true result with an absolute value that is less than the minimum representable value of the actual destination operand (e.g., MUL_SR_TR yielding a true result with an exponent less than -16,382).

- Inexact -- caused by a true result of a floating-point operation that is not exactly representable in the specified precision of the destination format (e.g., the true result of a CVT_TR_SR instruction is not exactly representable in the Short-Real destination). This fault only occurs when the Inexact Control bit is set in the current process status.

The Data Operator Faults section of Chapter 12, "Fault and Trace Reference," describes the specific conditions that cause these faults to occur for each GDP operator.

At the time a fault occurs, the GDP automatically places information in the Context Fault Data Area of the current process object. This information defines the type and circumstances surrounding the fault. The appropriate fault handler can then use this information to diagnose the fault and undertake repairs. Only those fields in the fault data area used to record data unique to the given fault are valid on entry to the fault handler.

Data operator faults are also classified according to whether they are Pre-operation or Post-operation. A pre-operation fault occurs at the start of instruction execution when the GDP examines the source operand(s). All Domain Error faults are pre-operation faults. A pre-operation fault causes the restoration of the operand stack pointer to its value at the start of the faulted instruction. In addition, the GDP places appropriate values in the following fault data fields:

- First Fault Data Item (containing source operand 1)

- Second Fault Data Item (containing source operand 2 when appropriate, according to the operator attempted)

The values placed in these fields are determined by the operator attempted. The operands stored into the fault data items are always justified into the least-significant portion of the field with any leftover upper bytes undefined.

A post-operation fault occurs after a faulted instruction that has been executed up to, but not including, storing a result. The operand stack pointer is not restored to its previous value. The GDP places the exceptional result value in the First Fault Data Item field. All overflow, underflow, and inexact faults are post-operation faults.

FLOATING-POINT FAULTING

Floating-point Domain Errors

Domain errors occur when an attempt is made to execute an operator with an operand value that is outside the domain that is defined for the attempted operation. These floating-point domain errors are defined:

- attempting to divide by ± zero.
- attempting to take the square root of a negative value.
- attempting any arithmetic, relational, or conversion operator with an invalid floating-point operand. The various kinds of invalid floating-point operands are discussed in earlier sections of this chapter.

Floating-Point Overflow

A floating-point overflow fault occurs whenever a floating-point operator produces a result with an exponent that is algebraically too large for the exponent field of the destination operand. Temporary-real overflow occurs if an instruction produces a temporary-real result with a true exponent greater than 16,383. Real overflow and short-real overflow can occur only during conversion instructions; real overflow occurs if the exponent of the source operand is greater than 1,023; short-real overflow occurs if the exponent of the source operand is greater than 127.

All overflow faults are post-operation faults. For a temporary-real overflow fault, the exceptional result written to the First Fault Data Item has the correct, properly rounded significand, but the exponent is "wrapped around" with a value that is 32,767 less than the true exponent. A similar kind of exceptional result could be written in the case of short-real or real overflow, but a wrapped-around exponent is not meaningful when overflow is caused by a conversion operator. Thus, for short-real and real overflows, the exceptional result written to the First Fault Data Item is the value of the source operand referenced by the conversion instruction causing the fault.

Floating-Point Underflow

A floating-point underflow occurs whenever a floating-point operator produces a result with an exponent that is algebraically too small for the exponent field of the destination operand. Temporary-real underflow occurs if an instruction produces a temporary-real result with a true exponent less than -16,382. Real and short-real underflow can occur only during conversion instructions; real underflow occurs if the exponent of the source operand is less than -1022; short-real underflow occurs if the exponent of the source operand is less than -126.

All underflow faults are post-operation faults. For a temporary-real underflow fault, the exceptional result written to the First Fault Data Item has the correct, properly rounded significand, but the exponent is "wrapped around" with a value that is 32,767 greater than the true exponent. For short-real and real underflows, the exceptional result written to the First Fault Data Item is the value of the source operand referenced by the conversion instruction causing the fault.

Floating-Point Inexact

An inexact fault occurs if the result of a floating-point operation is not exactly representable in the destination operand. Whether or not the representation can be accomplished exactly may depend on the current Precision Control. The setting of the Inexact Control bit in the GDP and the Process Status determines whether an inexact fault will occur. If the Inexact Control bit is one, then an instruction with an inexact result does not store its result and raises the inexact fault. The inexact fault is a post-operation fault. The value stored in the First Fault Data Item is the value of the exceptional temporary-real result before any rounding.

PART II

REFERENCE INFORMATION

CHAPTER 9
OBJECT SET

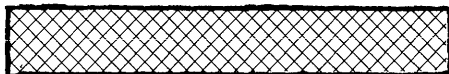
This chapter defines the Object Set of the iAPX 432 General Data Processor. The formats and processor interpretation of the fields are given for all system objects.

CHAPTER CONVENTIONS

Throughout this chapter, certain conventions are used. They are described in the following sections.

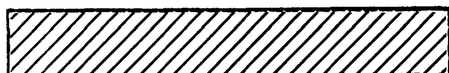
RESERVED FIELDS

Reserved fields are reserved for use by the processor. Software can write these fields but cannot depend on retrieving the value written. Reserved fields are indicated in the illustrations of this chapter using the following graphic convention:



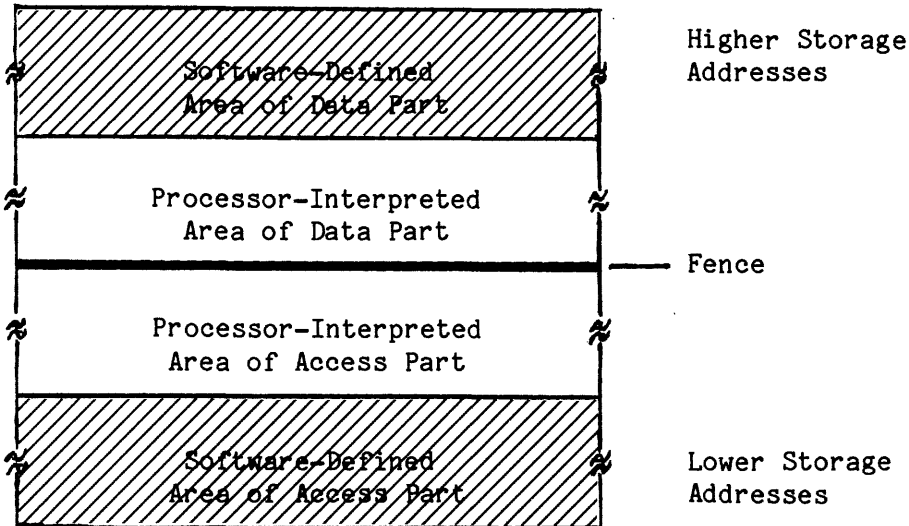
PRESERVED FIELDS

Preserved fields are not affected by the processor after their creation and may be used by software. Preserved fields are indicated in the illustrations of this chapter using the following graphic convention:



OBJECT ILLUSTRATION CONVENTION

System objects are shown in this chapter using the following graphic convention to illustrate the parts of each object:



The Processor-Interpreted areas contain the fields that uniquely characterize the system object to a processor. The Software-Defined areas are optional variable-sized portions of the object that are preserved by processors for exclusive use by software. Some system objects (e.g., port objects) cannot include Software-Defined areas. The Fence is the graphic convention used to indicate the boundary between the Access Part (AP) and the Data Part (DP) of an object.

ENCODED VALUES

Unless otherwise noted, binary encoded field values are shown in this chapter in an most-significant bit (MSB) to least-significant bit (LSB) order, left to right.

INDEX FIELDS

Values in index fields select an element from an array of entries or descriptors. Such values are multiplied by the length of a descriptor in bytes (16 bytes for OTEs and 4 bytes for ADs) to obtain the byte displacement into the array. The index value itself counts descriptors starting at zero. Unless otherwise noted, index field values are automatically scaled by the processor (i.e., multiplied by the appropriate descriptor length) to obtain byte displacements relative to the Fence of an object.

DISPLACEMENT FIELDS

In general, a displacement is a length (counting either bytes or bits) from the base (Fence) of an object to a specified point in the object. Unless otherwise noted, displacement fields in this chapter contain values that count bytes. Displacements are relative to the fence. AP displacements are negative offsets (i.e., from the fence toward lower physical addresses) and DP displacements are positive offsets (i.e., from the fence toward higher physical addresses).

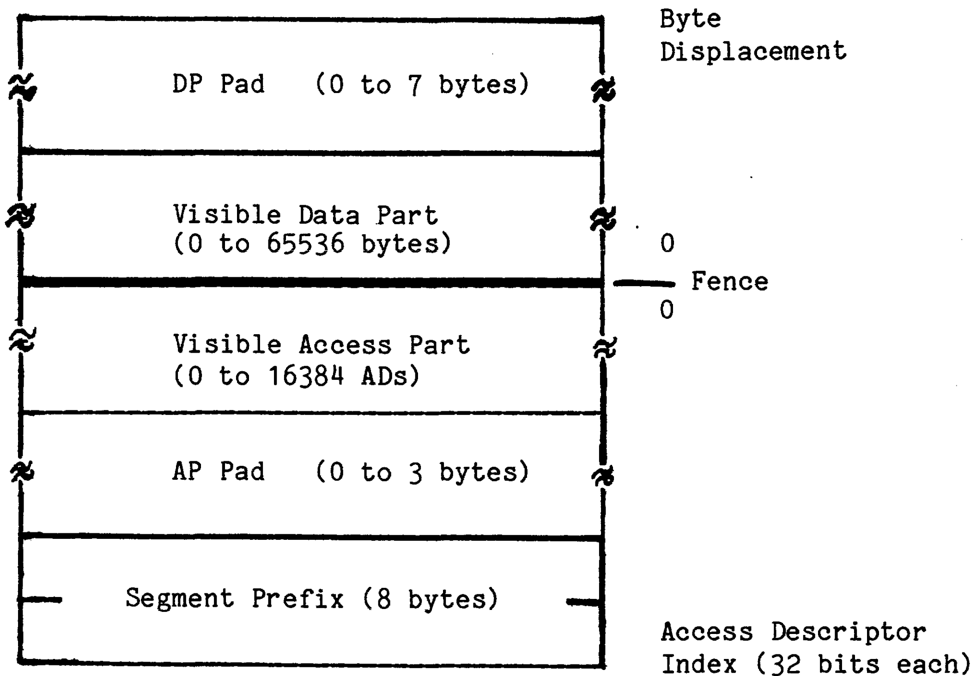
OBJECT REPRESENTATION

Objects are physically represented by segments. There are two address spaces in which segments can be defined: the storage address space and the interconnect address space. All system objects are defined in the storage address space. System objects consist of an access part and a data part, although one or the other may be nonexistent in a given system object. For example, an instruction object may have no access part.

The general storage segment structure is maintained by the processor and is normally transparent to the user. Therefore, throughout this manual, only the visible segment part of the general storage segment is shown.

GENERAL STORAGE SEGMENT STRUCTURE

The format of a general storage segment is as follows.



Storage segments are always aligned on double-word (8-byte) boundaries. The first 4 bytes of the 8-byte segment prefix contain an image of the original access descriptor to the visible segment. These first 4 bytes of the segment are first in terms of absolute physical storage address. The segment AD image is initially written by the processor (as a valid AD with all rights) when the segment is created and is preserved thereafter. The second 4 bytes of the segment prefix are preserved. The segment prefix is not normally accessible. There are cases, however, in which operating system software uses the segment prefix during storage management. For example, the segment AD image is used by the parallel storage compaction process.

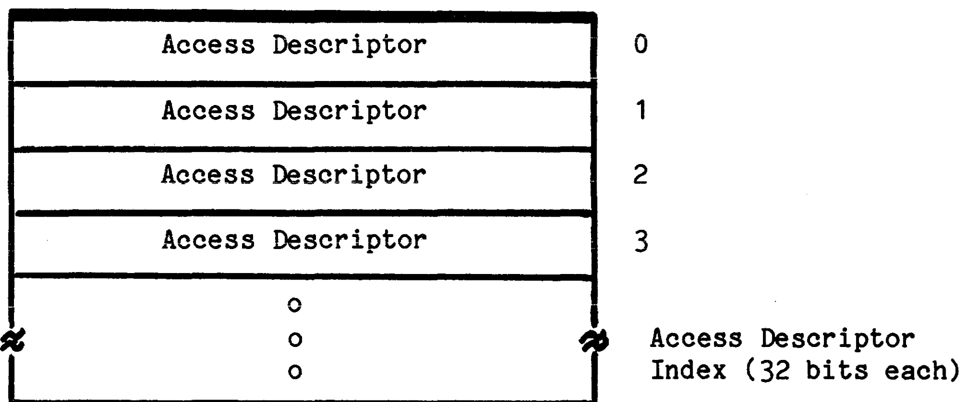
The maximum visible segment size is 131,072 bytes, while the maximum size of each visible part is 65,536 bytes. This does not include the segment prefix. The DP pad is a 0- to 7-byte area that is added to the end of the visible segment to round the size of the segment up to the nearest 8-byte integer multiple. The AP pad is a 0- or 4-byte area that is added between the visible access part and the segment prefix. These pads ensure the alignment of segments on 8-byte boundaries and also minimize fragmentation of physical storage. The rounding in size is required by the processor and is performed automatically if the segment is created by the GDP.

The Fence is the imaginary boundary separating the access and data parts.

For objects in the Interconnect Address Space, the General Interconnect Segment Structure contains a Visible Data Part only, no pads, and no Segment Prefix. It must be aligned on a double-byte boundary and must have an even length.

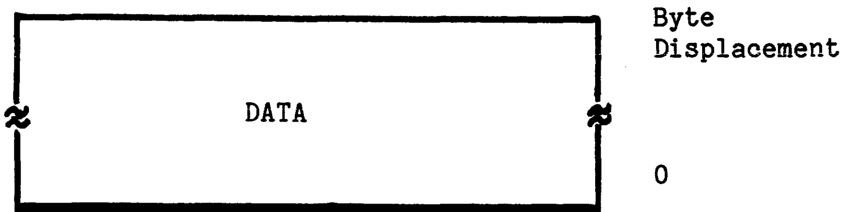
ACCESS PART

The general format of an Access Part (AP) is shown below. An access part consists of an array of access descriptors. Access parts can contain only access descriptors. Access descriptors are laid out in memory on an AD by AD basis in decreasing physical addresses, and the least-significant bit of each AD occupies the lowest physical address. Access descriptors are only interpreted as such by the processor when they reside in an access part. A copy of an AD in a data part, object descriptor, or segment prefix is called an access descriptor image and can neither be used as an AD directly nor copied into an AD location in an access part.



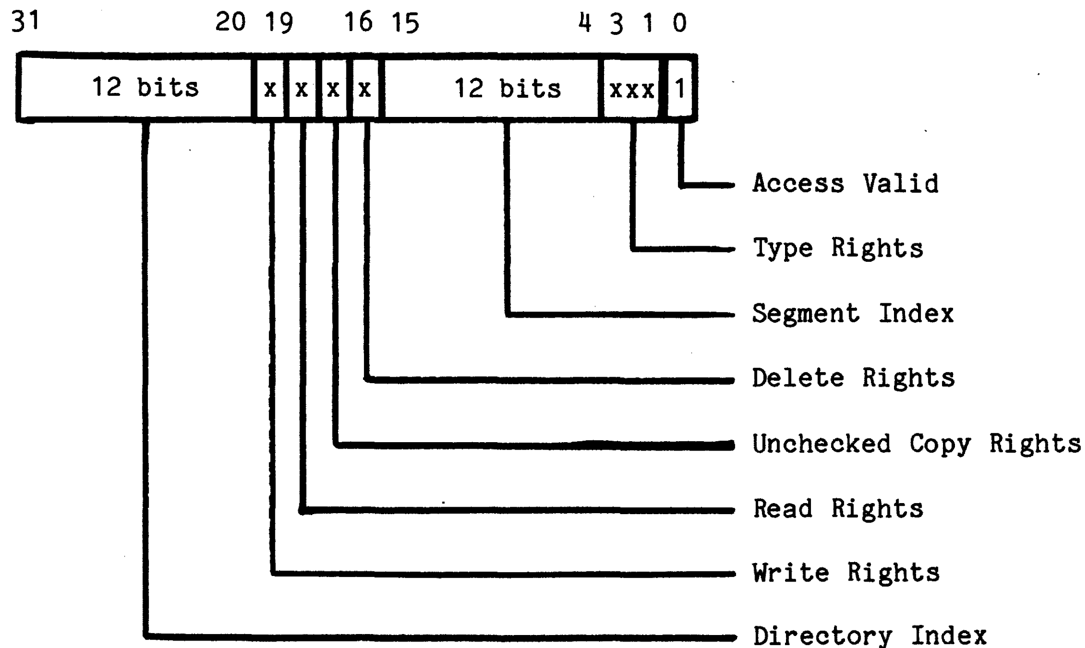
DATA PART

A Data Part (DP) contains ordinary data. Many system objects have a data part containing status and control information required for that object. Generic data parts can contain any programmer-defined data structures and operands. Data parts are depicted in this chapter with the following general form.



ACCESS DESCRIPTOR

Access Descriptors are the primary means of object addressing in the iAPX 432 architecture. They consist of the following fields:



The fields that consitute an access descriptor are interpreted as follows:

Access Valid (Bit 0)
If access valid is 0, this access descriptor is interpreted as null (i.e., invalid for object addressing). The remaining bits are not interpreted by the processor, but can be used for a 31-bit embedded data value (described in the next section of this chapter).

Type Rights (Bits 1 - 3)
The interpretation of this 3-bit field is determined by the object type of the referenced object. These bits are called:
Bit 1 - Type Right 1
Bit 2 - Type Right 2
Bit 3 - Type Right 3
The individual system object descriptions in later sections of this chapter describe the interpretation of the type rights field in access descriptors that reference system objects. For many system objects, some or all of these bits are uninterpreted (preserved). Type Rights bits that are uninterpreted by the processor can be used by software to define additional rights for objects of a particular type.

Segment Index (Bits 4 - 15)
This 12-bit field contains the index into a selected object table of the object descriptor for the object referenced by this access descriptor. The object table itself is selected using the Directory Index field described below.

Delete Rights (Bit 16)

This bit indicates whether this access descriptor can be deleted (i.e., can be overwritten). If delete rights is 0, and an attempt is made to delete this valid access descriptor, an Access Descriptor Deletion Fault occurs. If the bit is 1, deletion can occur without faulting. Whenever an access descriptor is copied, the delete rights bit of the copy is set to 1, so that it may later be deleted; otherwise, a proliferation of undeletable access descriptors might occur. The Delete Rights bit is not interpreted in null ADs. Thus, null ADs may always be overwritten.

Unchecked Copy Rights (Bit 17)

This bit indicates whether a level compatibility check is required when this access descriptor is copied. If this bit is 1, the level compatibility check is bypassed. Setting this bit to 1 via an AMPLIFY RIGHTS instruction should be done with extreme caution as it may result in one or more ADs for a previously deleted object.

Read Rights (Bit 18)

This bit indicates whether the access descriptor can be used to read from the object it references. If read rights is 1, the access descriptor can be used to read from the referenced object.

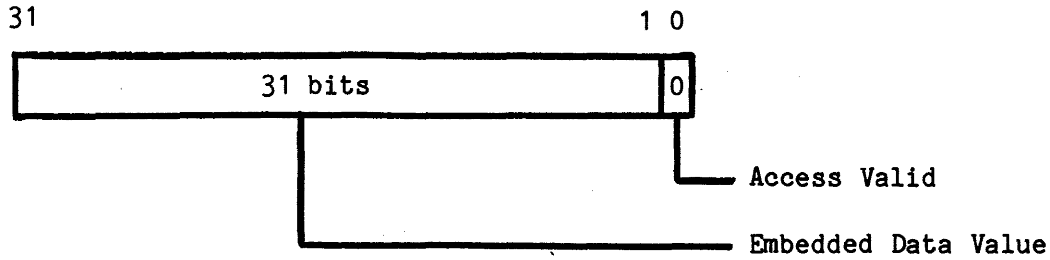
Write Rights (Bit 19)

This bit indicates whether the access descriptor can be used to write to the object it references. If write rights is 1, the access descriptor can be used to write to the referenced object.

Directory Index (Bits 20 - 31)

This 12-bit field contains an index into the object table directory. It thus yields a storage descriptor containing the base address and length of an object table. The selected object table is indexed by the 12-bit Segment Index field (described above) to select the object descriptor for the object referenced by this access descriptor.

EMBEDDED DATA VALUE



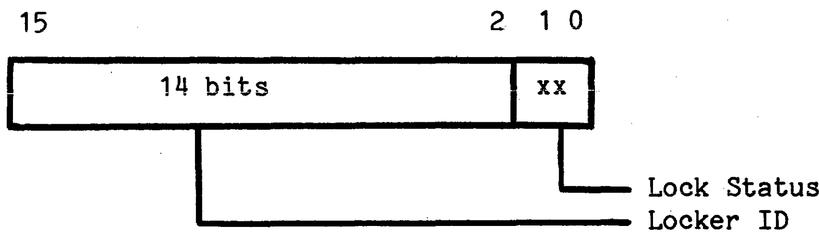
The fields that consitute a null AD can be interpreted by certain operators as follows:

Access Valid (Bit 0)
For embedded data values the access valid bit is 0 and thus the AD in which the EDV is embedded is null (i.e., invalid for object addressing). The remaining bits are not interpreted by the processor and can be used for a 31-bit embedded data value.

Embedded Data Value (Bit 1 - 31)
This field contains a 31-bit value that can be passed by value between processes or contexts like an AD for a message or parameter, without the overhead in memory space or access time of referencing the value within an object.

OBJECT LOCK

Object locks must always be located in the data part of objects. The processor recognizes object locks at specific locations in system objects and automatically manipulates them during normal operations to accomplish mutually exclusive access among contending processors and processes. See the specific system object descriptions in this chapter for the exclusive access coverage defined for the object lock in that system object. Object locks can be manipulated by software via LOCK OBJECT and UNLOCK OBJECT instructions. In this case, long-term software locking is the lock status used. Mutual exclusion is then accomplished only if all contending processes honor the convention explicitly. An object lock consists of the following fields:



The fields that constitute an object lock are interpreted as follows:

Lock Status (Bits 0 - 1)

Lock status values are interpreted as follows:

00 - Not locked
01 - Hardware locked
10 - Long-term software locked
11 - Short-term software locked

Hardware locking is set by the processor when performing an operation on behalf of a processor that requires the object to be locked.

Short-term software locking is set by the processor when executing an instruction on behalf of a process that requires the object to be locked for the duration of one instruction.

Long-term software locking is set by the processor when a software operation has specified (via a LOCK OBJECT instruction) that an object be locked. A long-term software lock remains in effect until an UNLOCK OBJECT instruction is performed on the object.

Locker ID (Bits 2 - 15)

If hardware-locked, this field is written by the processor. (Bits 8 - 15 contain the left-justified 8-bit processor ID of the locking processor, and bits 2 - 7 contain zeros). The processor ID must be nonzero. If software locked, this field (bits 2 - 15) is written by the processor to contain the process ID (from the process data part of the locking process). The process ID must be nonzero.

OBJECT DESCRIPTIONS

SYSTEM OBJECT TYPES

Following sections of this chapter contain descriptions of the system objects interpreted by the 432 architecture. The following table gives the order in which the system objects are presented and the page on which each description begins.

System Object Type	Page
Object Table Object	9-12
Processor Object	9-27
Processor Communication Object	9-33
Process Object	9-35
Context Object	9-43
Domain Object	9-48
Instruction Object	9-49
Port Object	9-51
Carrier Object	9-55
Storage Resource Object	9-59
Storage Claim Object	9-61
Physical Storage Object	9-62
Type Definition Object	9-65
Dynamic Type Object	9-66
Type Control Object	9-67

Table of System Object Types and Their Type Rights

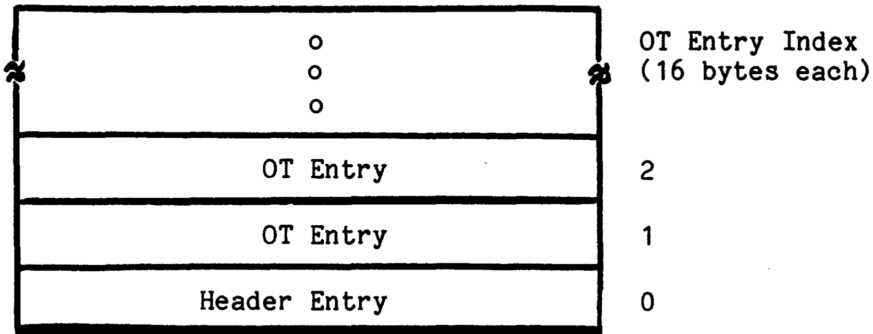
The following table lists all system object types and any type rights interpreted by the processor in ADs that reference the system object.

System Object Type	Type Right 3	Type Right 2	Type Right 1
Object Table	----	----	----
Processor	----	----	----
Processor Communication	----	----	Send IPC
Process	----	----	Set Mode
Context	----	----	Return
Domain	----	----	----
Instruction	----	Trace	----
Port	Send Process	Receive	Send
Carrier	----	----	Surrogate
Storage Resource	----	----	Create
Storage Claim	----	----	----
Physical Storage	----	----	----
Type Definition	----	----	----
Dynamic Type	----	----	----
Type Control	Refine	Amplify	Create

---- indicates that the type right is not interpreted by the GDP (and is preserved).

OBJECT TABLE OBJECT

An object table object contains an array of object table entries (OTEs), each of which is 16 bytes in length. Refinement descriptors with object table as their object type are not supported by the GDP.



OBJECT TABLE ENTRIES

An object table can contain the following types of object table entries. Each is described in its own section later in this chapter.

Header Entry
: Each header entry is used by the processor to control storage allocation using the object table. The first entry in an object table can only be a header entry.

Free Entry
: Each OT free entry is a place holder for a potential object descriptor.

The following object table entries are called object descriptors (ODs):

Storage Descriptor
: A storage descriptor defines a object allocated in the storage address space.

Refinement Descriptor
: A refinement descriptor defines an object consisting of a restricted view of parts of a previously defined storage segment.

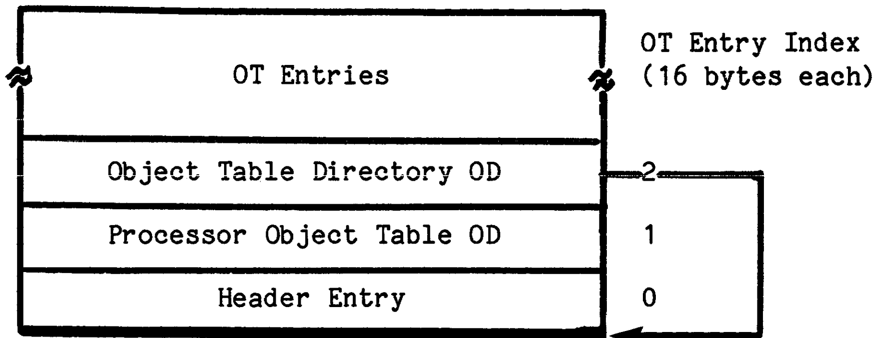
Interconnect Descriptor
: An interconnect descriptor defines a object in the interconnect address space.

Specific object table entries are identified by the lower 5 bits of each 16-byte entry as follows:

Bits 4:3	Bit 2	Bits 1:0	Object Table Entry
00	0	00	Free Entry
00	1	00	Header Entry
01	x	00	Interconnect Descriptor
xx	x	01	Reserved
xx	x	10	Refinement Descriptor
xx	x	11	Storage Descriptor

x means: Not used to identify the specific OTE (but may be used within another field within the OTE).

OBJECT TABLE DIRECTORY



An object table directory (OTD) is a special kind of object table object that contains only ODs describing object table objects. Only the first three entries are uniquely processor-interpreted for an object table directory. They are:

Header Entry (OTE 0)
This is an ordinary header entry, as described in its own separate section of this chapter.

Processor Object Table OD (OTE 1)
This storage descriptor defines the object table containing object descriptors for processor objects. During processor qualification, a processor indexes into the processor object table with its 8-bit processor ID to find the object descriptor for its processor object.

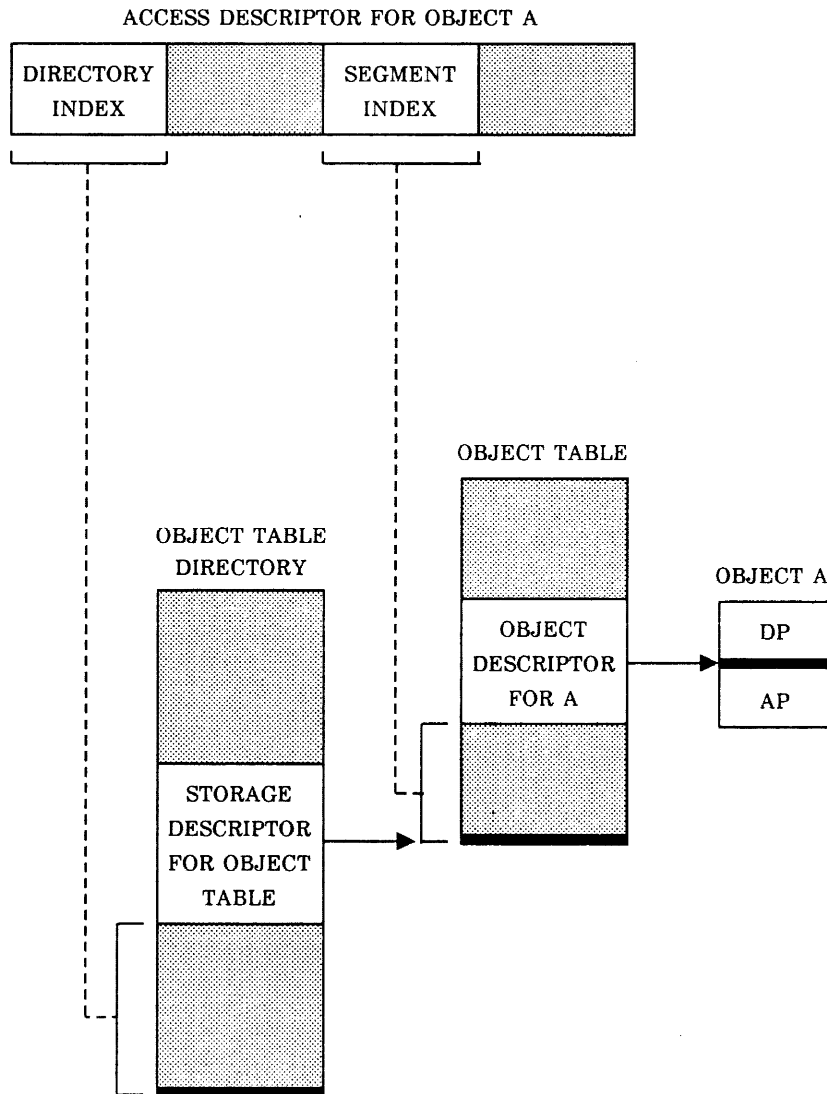
Object Table Directory OD (OTE 2)
This storage descriptor defines the object that is this object table directory itself. All OTDs must have an OD for themselves at OTE index 2.

TYPE RIGHTS FOR OBJECT TABLE OBJECTS

The type rights in an access descriptor that references an object table object are uninterpreted.

OBJECT ADDRESSING SUMMARY

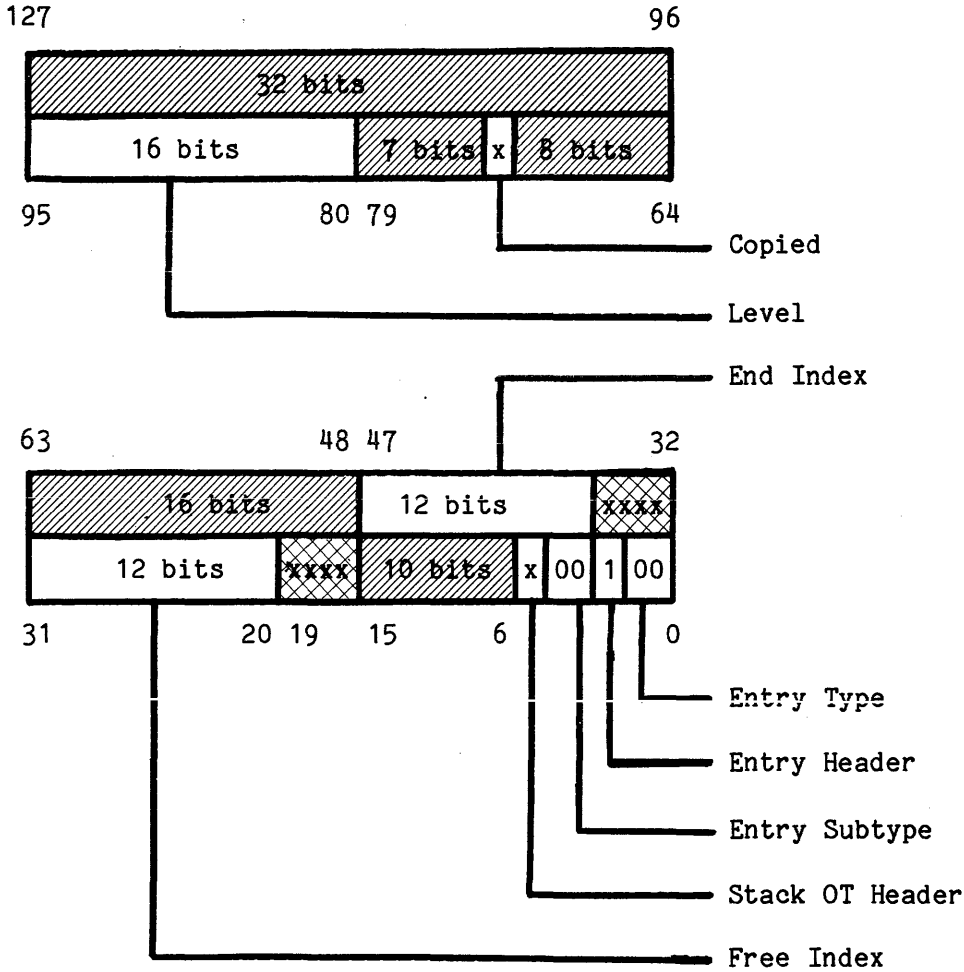
The following diagram summarizes object addressing in the iAPX 432 architecture. The Object Descriptor for A is often called the access descriptor's associated object descriptor.



OBJECT TABLE ENTRY DESCRIPTIONS

The following sections of this chapter contain descriptions of the types of object table entry.

HEADER ENTRY



The fields that constitute a header entry are interpreted as follows:

Entry Type (Bits 0 - 1)
    This field is 00 for a header entry.

Entry Header (Bit 2)
    This bit is 1 for a header entry.

Entry Subtype (Bits 3 - 4)
    This field is 00 for a header entry.

Stack OT Header (Bit 5)
This bit is 0 for heap object table headers. The free entries associated with a heap OT header entry are organized as a linked list starting with the free entry indexed by the free index field. If the free index field is zero in a heap OT header, then there are no free entries associated with the header (the list is empty).
This bit is 1 for stack object table headers. The free entries associated with a stack OT header entry are from the free index + 1 to the end index (inclusive). If the free index is greater than or equal to the end index field in a stack OT header, then there are no free entries associated with the header.

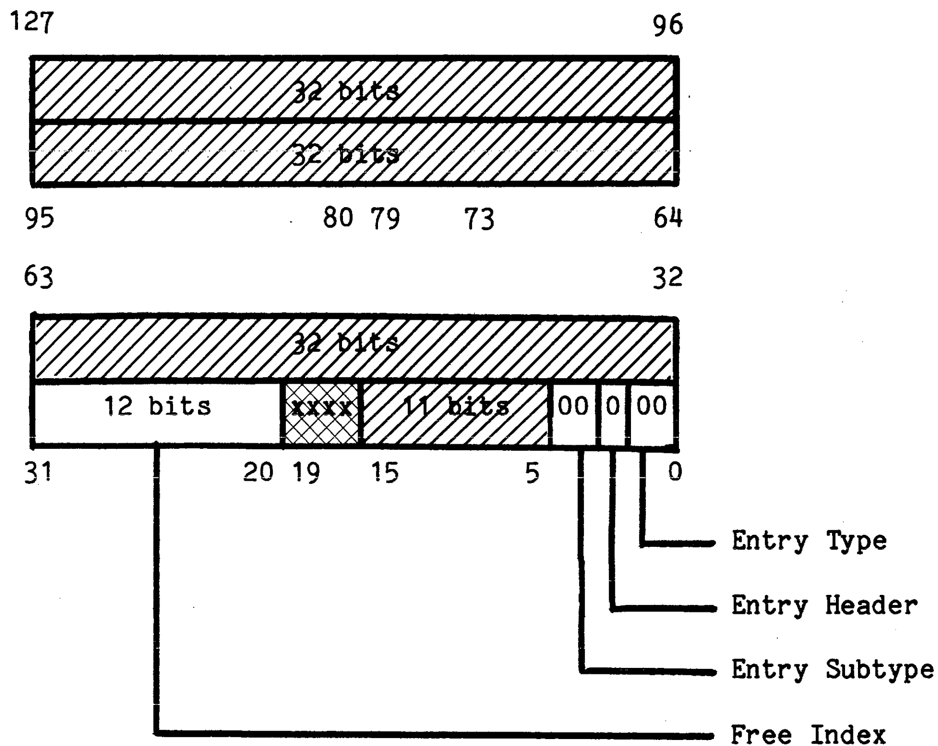
Free Index (Bits 20 - 31)
If the Stack OT Header bit is 0, for heap allocation, all free entries (described later in this chapter) are in a linked list. The free index field in the header entry indexes the first free entry (if any) in the list. If the Stack OT Header bit is 1, for stack allocation, the free index field indexes the most recently allocated object descriptor in the object table, and free index + 1 indexes the first free entry (if any).

End Index (Bits 36 - 47)
This 12-bit field is interpreted only if the Stack OT Header bit is 1 (for stack object tables). It then indexes the last object table entry in the object table. When Free Index >= End Index, all OTEs in the stack object table have been allocated. Otherwise, the OTEs from Free Index+1 through End Index are free entries.

Copied (Bit 72)
This bit indicates whether an access descriptor that references this object table header has been copied since this bit was last set to 0 by software. The copied bit is initialized to 1 when this object descriptor is created. Furthermore, the copied bit is set to 1 by the processor whenever an AD that references this entry is copied. The Copied bit serves as the gray bit for the iMAX parallel garbage collector algorithm.

Level (Bits 80 - 95)
This 16-bit field contains either the value found in the Current Allocation Level field of the process object (during stack allocation) or the Allocation Level field of the SRO (during heap allocation) at the time this object descriptor was allocated. For an OD allocated with a level of 0, its associated AD has the Unchecked Copy Rights bit set to 1 when created.

FREE ENTRY



Free Entries are only interpreted as such by the processor in heap object tables. The fields that constitute a free entry are interpreted as follows:

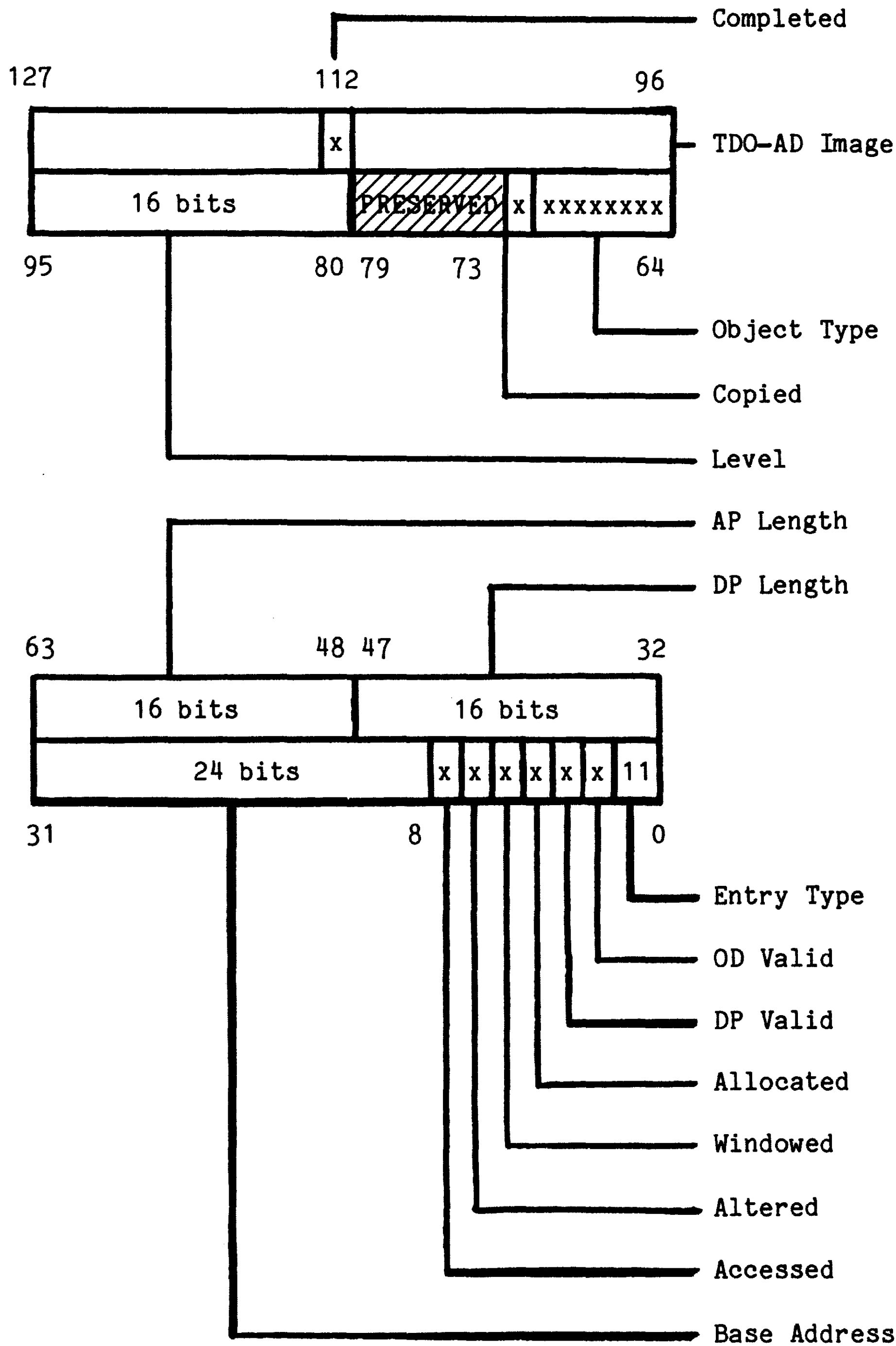
Entry Type (Bits 0 - 1)
This field is 00 for a free entry.

Entry Header (Bit 2)
This bit is 0 for a free entry.

Entry Subtype (Bits 3 - 4)
This field is 00 for a free entry.

Free Index (Bits 20 - 31)
This field is only interpreted for heap object tables. The field indexes from the base (fence) of this object table to the next free entry in the linked list of free entries. The last such free entry in the list is identified by a Free Index value of zero.

STORAGE DESCRIPTOR



The fields that constitute a storage descriptor are interpreted as follows:

Entry Type (Bits 0 - 1)
This field is 11 for a storage descriptor.

OD Valid (Bit 2)

If this bit is 0, only the Copied bit and Entry Type fields have meaning in this descriptor. A fault occurs if OD Valid is 0 and an attempt is made to reference a object through this object descriptor.

DP Valid (Bit 3)

If this bit is 0, the object defined by this storage descriptor does not have a data part. Otherwise, the size of the data part is specified by the DP Length field.

Allocated (Bit 4)

This bit is 0 if there is no storage allocated with this storage descriptor, and 1 if there is storage allocated with it. If this bit is 0, the base address field in this descriptor is undefined. Each time the object defined by this descriptor is accessed (i.e., read from or written to), this bit is checked. If it is 0 and the object defined by this descriptor is not qualified within the GDP, a fault occurs.

Windowed (Bit 5)

This bit is referenced by iAPX 432 Interface Processors (IPs) and by GDP software to determine if the object described by this storage descriptor is being mapped by an IP window. If windowed is 1, then an IP window is open on all or part of the object. This bit is not interpreted by the GDP and is initially 0. This bit is set and cleared by the IP operations that open and close windows.

Altered (Bit 6)

This bit is initialized to 1 to denote that the object defined by this storage descriptor has been altered (i.e., has been written into). This bit is set to 1 by the processor whenever any portion of the object is overwritten. It is cleared by operating system software in a virtual memory system. In a non-virtual memory system, this bit should be left as 1 by software to avoid unnecessary but automatically occurring storage accesses.

Accessed (Bit 7)

This bit is initialized to 1. Subsequently, if the object defined by this storage descriptor is accessed (i.e., read from or written to), this field is set to 1 by the processor. It is cleared by software in a virtual memory system. In a non-virtual memory system, this bit should be left as 1 by software to avoid unnecessary but automatically occurring storage accesses.

Base Address (Bits 8 - 31)

This 24-bit field contains the physical base address (in bytes in the storage address space) of the object. This value is the address of the first byte in the data part of the object. It is also the address of the first byte above (i.e., at the next higher address) the first AD in the access part of the object.

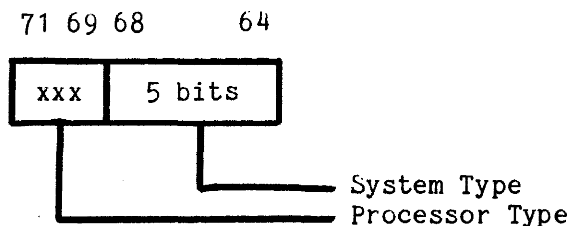
DP Length (Bits 32 - 47)

The value of this 16-bit field is one less than the length in bytes of the data part of the defined object. Thus, a maximum-length DP of 65,536 bytes has as its DP Length field a value of 65,535 (0FFFFH). Each time a operand is referenced by a logical address

in an instruction, the operand offset is compared to this DP Length field. If (operand offset + operand size) exceeds the actual length in bytes of the data part, a fault occurs.

AP Length (Bits 48 - 63)
The value of this 16-bit field is one less than the length in bytes of the access part of the defined object. Thus, a maximum-length access part of 65,536 bytes has as its AP Length field a value of 65,535 (0FFFFH). Each time an object is referenced in the course of generating an address using a scaled AD index, the displacement into the AP is compared to this AP Length field. If the displacement exceeds the AP length, a fault occurs. Note that AP length values of 0, 1, or 2 indicate that the object has no access part.

Object Type (Bits 64 - 71)
This 8-bit field encodes the object type of the object. An object type is composed of a 5-bit System Type field and a 3-bit Processor Type field:



System Type (Bits 64 - 68)
This 5-bit field determines the system type of the object defined by the storage descriptor. The encodings for the System Type field are as follows:

Encoding	System Type
00000	Generic Object
00001	Object Table Object
00010	Domain Object
00011	Instruction Object
00100	Context Object
00101	Process Object
00110	Processor Object
00111	Port Object
01000	Carrier Object
01001	Storage Resource Object
01010	Physical Storage Object
01011	Storage Claim Object
01100	Dynamic Type Object
01101	Type Definition Object
01110	Type Control Object
01111	RESERVED
10000	Processor Communication Object
10001 thru 11111	RESERVED

Processor Type (Bits 69 - 71)
This 3-bit field encodes the type of iAPX 432 processor for which the object is defined. If a processor attempts to access an object that is not of its type, a fault occurs. The encodings for the Processor Type field are as follows:

Encoding	Processor Type
000	All
001	GDP
010	IP
011 thru 111	Reserved

GDPs can only read or write objects with processor type All or GDP.

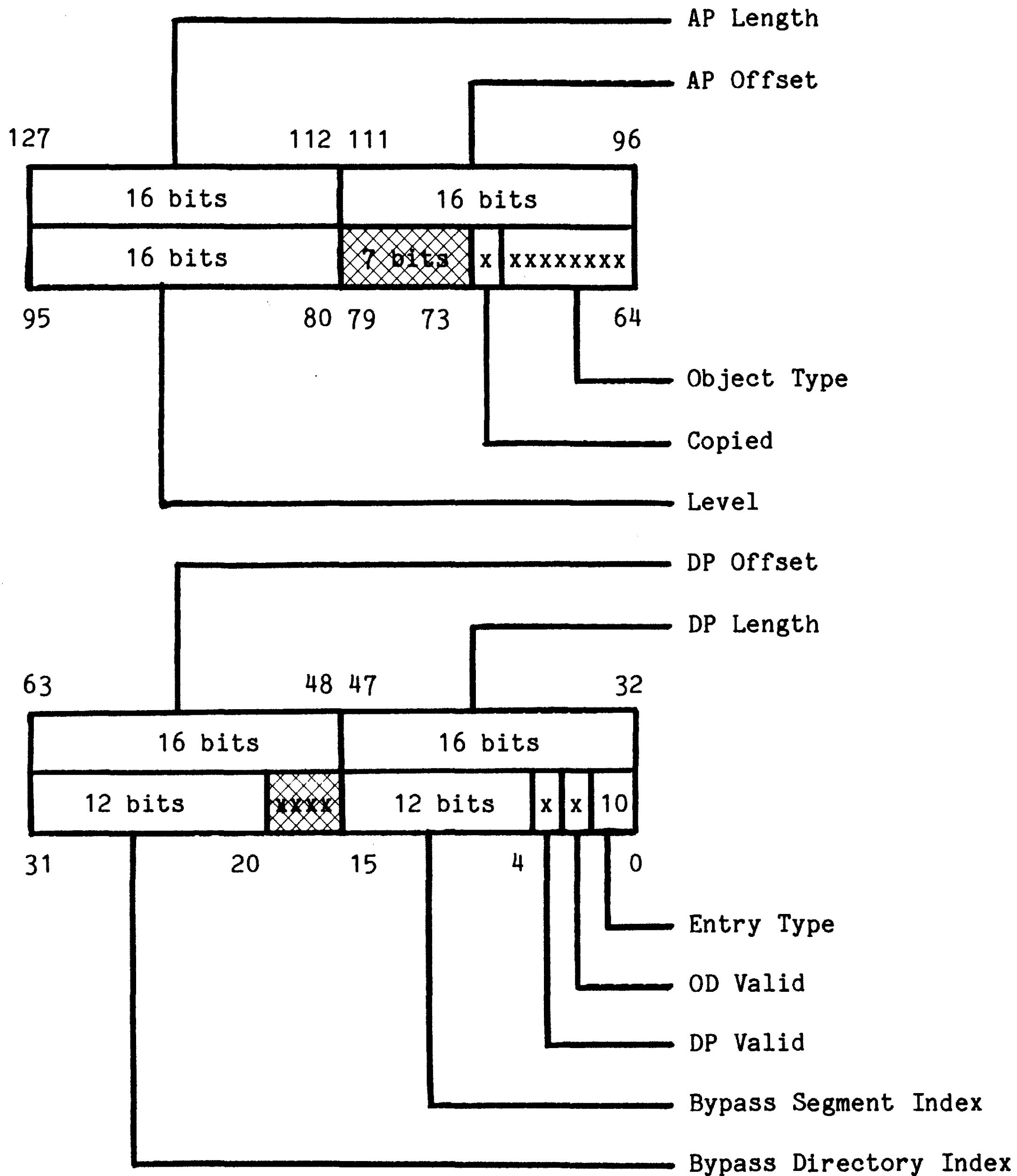
Copied (Bit 72)
This bit indicates whether an access descriptor that references this object descriptor has been copied since this bit was last set to 0 by software. The copied bit is initialized to 1 when this object descriptor is created. Furthermore, the copied bit is set to 1 by the processor whenever an AD is copied which references this OD. The Copied bit serves as the gray bit for the iMAX garbage collector algorithm.

Level (Bits 80 - 95)
This 16-bit field contains the value found in either the Current Allocation Level field of the process object (during stack allocation) or the Allocation Level field of the SRO (during heap allocation) at the time this object descriptor was allocated. For an OD allocated with a level of 0, its associated AD has the Unchecked Copy Rights bit set to 1 when created.

TDO-AD Image (Bits 96 - 127)
This 32 bit field contains an image of an access descriptor for the Type Definition Object that defines the type manager of the object instance described by this storage descriptor.

Completed (Bit 112)
The interpretation of this bit is only important during the creation of the object defined by this object descriptor. System software can consult the Completed bit to determine if segment creation is still in progress. If the Completed bit is 1, the segment creation process has initialized the new segment to all zeros (and all ADs in the new segment to all null ADs with EDVs zero). This bit corresponds to the position of the Delete Rights bit in the TDO-AD Image. But, the TDO-AD Image can never be used in-place as an AD. When copied into an access part by a processor, the Delete Rights bit in the AD image is always set to 1.

REFINEMENT DESCRIPTOR



The fields that constitute a refinement descriptor are interpreted as follows:

Entry Type (Bits 0 - 1)
This field is 10 for a refinement descriptor.

OD Valid (Bit 2)
If this bit is 0, only the Copied bit and Entry Type fields have meaning in this descriptor. A fault occurs if OD Valid is 0 and an attempt is made to reference an object through this object descriptor.

DP Valid (Bit 3)
If this bit is 0, the object defined by this object descriptor does not have a data part. Otherwise, the size of the data part is specified by the DP Length field.

Bypass Segment Index (Bits 4 - 15)
This 12-bit field contains the index into the selected object table to the storage descriptor for the underlying object. The object table itself is selected using the bypass directory index described below.

Bypass Directory Index (Bits 20 - 31)
This 12-bit field contains an index into the object table directory. It thus yields a storage descriptor containing the base (fence) address of an object table. The selected object table is indexed by the 12-bit bypass segment index (described above) to select the storage descriptor of the underlying object.

DP Length (Bits 32 - 47)
This 16-bit field contains a value that is one less than the length, in bytes, of the data part of the refinement.

DP Offset (Bits 48 - 63)
This 16-bit field contains a byte offset that is added to the base (fence) address of the underlying object to form the "imaginary" base address of the data part of this refinement.

Object Type (Bits 64 - 71)
This 8-bit field encodes the object type of this refinement. This field is interpreted the same as the object type field in a storage descriptor (described earlier in this chapter).

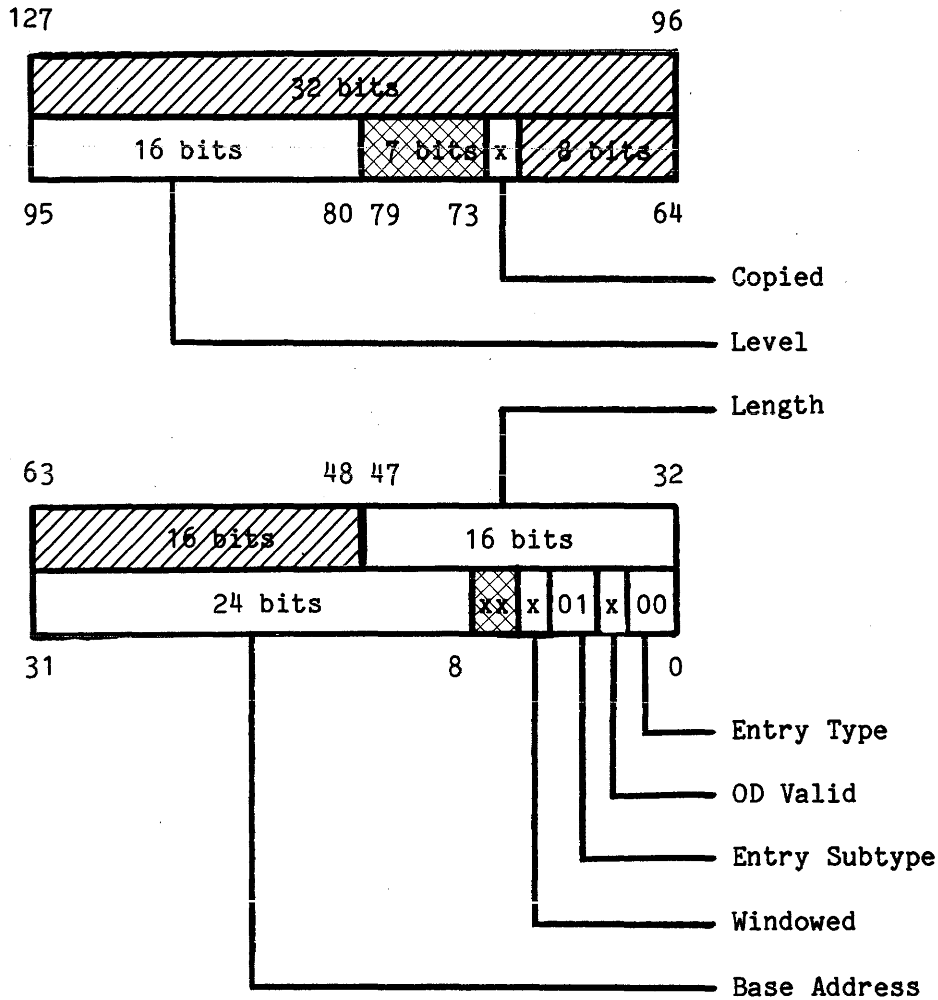
Copied (Bit 72)
This bit indicates whether an access descriptor that references through this object descriptor has been copied since this bit was last set to 0 by software. The copied bit is initialized to 1 when this object descriptor is created. Furthermore, the copied bit is set to 1 by the processor whenever an AD that references this OD is copied. The Copied bit serves as the gray bit for the iMAX parallel garbage collection algorithm.

Level (Bits 80 - 95)
This 16-bit field contains the value found in either the Current Allocation Level field of the process object (during stack allocation) or the Allocation Level field of the SRO (during heap allocation) at the time this object descriptor was allocated. For an OD allocated with a level of 0, its associated AD has the Unchecked Copy Rights bit set to 1 when created.

AP Offset (Bits 96 - 111)
This 16-bit field contains a byte offset that is subtracted from the base (fence) address of the underlying object to form the "imaginary" base address of the access part of this refinement.

AP Length (Bits 112 - 127)
This 16-bit field contains a value that is one less than the length, in bytes, of the access part of the refinement.

INTERCONNECT DESCRIPTOR



The fields that constitute an interconnect descriptor are interpreted as follows:

Entry Type (Bits 0 - 1)
This field is 00 for an interconnect descriptor.

OD Valid (Bit 2)
If this bit is 0, only the Copied bit and Entry Type fields have meaning in this descriptor. A fault occurs if OD Valid is 0 and an attempt is made to reference a object through this object descriptor.

Entry Subtype (Bits 3 - 4)
This field is 01 for an interconnect descriptor.

Windowed (Bit 5)
This bit is referenced by iAPX 432 Interface Processors (IPs) and by GDP software to determine if the object defined by the interconnect descriptor is being mapped by an IP window. If windowed is 1, then an IP window is open on all or part of the object. This bit is not interpreted by the GDP. This bit is set and cleared by the IP operations that open and close windows.

Base Address (Bits 8 - 31)
This 24-bit field contains the physical address (in bytes in the interconnect address space) of the interconnect object defined by this interconnect descriptor. This value is the address of the first byte of the object and must be even.

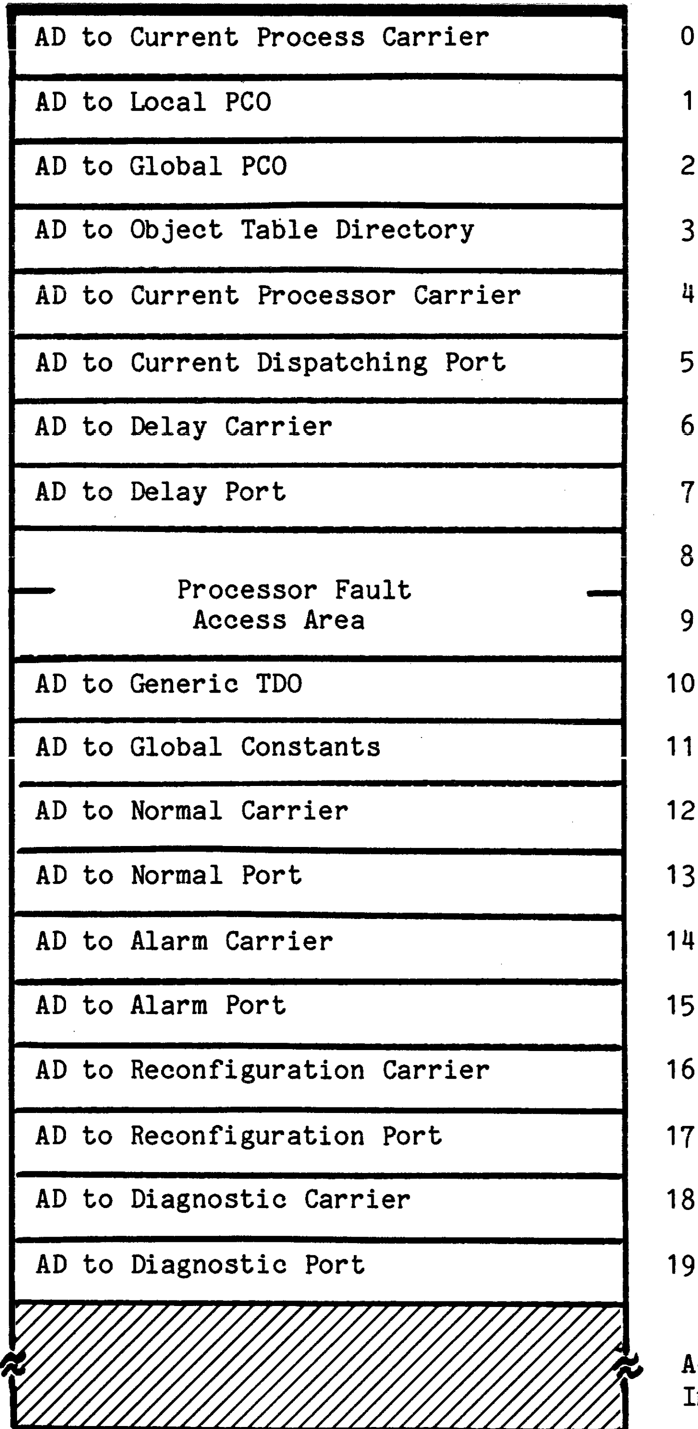
Length (Bits 32 - 47)
The value of this 16-bit field is one less than the length, in bytes, of the object defined by this interconnect descriptor.

Copied (Bit 72)
This bit indicates whether an access descriptor that references this object descriptor has been copied since this bit was last set to 0 by software. The Copied bit is initialized to 1 when this object descriptor is created. Furthermore, the Copied bit is set to 1 by the processor whenever an AD that references this OD is copied. The Copied bit serves as the gray bit for the iMAX parallel garbage collection algorithm.

Level (Bits 80 - 95)
This 16-bit field contains the value found in either the Current Allocation Level field of the process object (during stack allocation) or the Allocation Level field of the SRO (during heap allocation) at the time this object descriptor was allocated. For an OD allocated with a level of 0, its associated AD has the Unchecked Copy Rights bit set to 1 when created.

PROCESSOR OBJECT

PROCESSOR OBJECT (ACCESS PART)



Refinement descriptors with processor object as their object type are not supported by the GDP. The access descriptors that constitute the processor-interpreted access part of a processor object are interpreted as follows:

Current Process Carrier (AD 0)
This AD references the process carrier of the currently executing process of this processor.

Local PCO (AD 1)
This AD references the processor communication object used for local interprocessor communication.

Global PCO (AD 2)
This AD references the processor communication object used for global interprocessor communication.

Object Table Directory (AD 3)
This AD references the object table directory for this processor.

Current Processor Carrier (AD 4)
This AD is a copy of the AD that references the Normal, Alarm, Reconfiguration, or Diagnostic Carrier depending on the current dispatching mode.

Current Dispatching Port (AD 5)
This AD is a copy of the AD that references the Normal, Alarm, Reconfiguration, or Diagnostic Port depending on the current dispatching mode.

Delay Carrier (AD 6)
This AD references the delay carrier used by this processor to service the delay port.

Delay Port (AD 7)
This AD references the delay port used to provide the delay service for this processor.

Processor Fault Access Area (ADs 8 - 9)
These 2 ADs are written by the processor after a processor level fault and can be used by fault handling software. They are described in the Fault and Trace Reference chapter of this manual.

Generic TDO (AD 10)
This AD references a type definition object for generic objects.

Global Constants (AD 11)
This AD references a system wide Global Constants object. This AD must be the same as in the same-named fields of all processor objects and context objects.

The following 8 ADs (12 - 19) reference processor carriers and dispatching ports used by this processor. The current processor carrier and current dispatching port (each referenced by one of these ADs) are determined by the dispatching mode in the Processor Status field of the processor object. A copy of the current processor carrier AD occupies the AD 4 location of this processor object. A copy of the current dispatching port AD occupies the AD 5 location of this processor object.

Normal Carrier (AD 12)
This AD references the processor carrier used by a processor to receive or wait for a process at a normal port.

Normal Port (AD 13)
This AD references the dispatching port where this processor receives or waits for a normal process.

Alarm Carrier (AD 14)
This AD references the processor carrier used by a processor to receive or wait for a process at an alarm port.

Alarm Port (AD 15)
This AD references the alarm dispatching port where this processor receives or waits for a special alarm process.

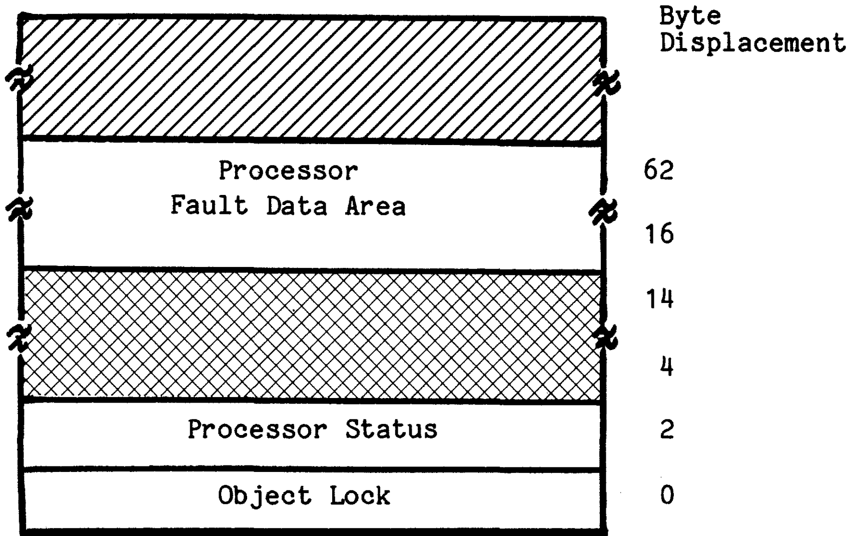
Reconfiguration Carrier (AD 16)
This AD references the processor carrier used by a processor to receive or wait for a process at a reconfiguration port.

Reconfiguration Port (AD 17)
This AD references the reconfiguration dispatching port where this processor receives or waits for a special reconfiguration process.

Diagnostic Carrier (AD 18)
This AD references the processor carrier used by a processor to receive or wait for a process at a diagnostic port.

Diagnostic Port (AD 19)
This AD references the diagnostic dispatching port where this processor receives or waits for a special diagnostic process.

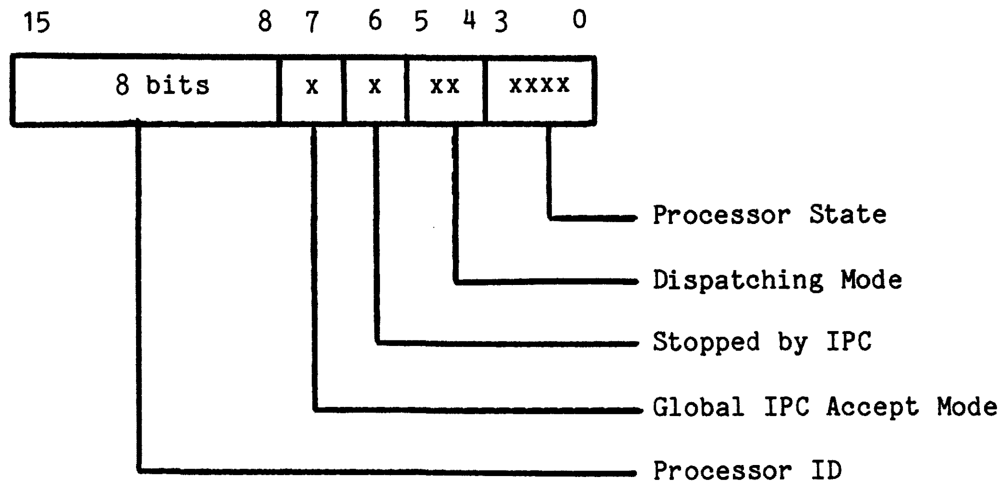
PROCESSOR OBJECT (DATA PART)



The fields that constitute the processor-interpreted data part of a processor object are interpreted as follows:

Object Lock (Bytes 0 - 1)
This field provides mutually exclusive access to this processor object and to its associated processor carriers and delay carrier. The object lock field is defined for many system objects and is described in the first part of this chapter.

Processor Status (Bytes 2 - 3)
The format of the processor status field is shown below:



The fields that constitute processor status are interpreted as follows:

Processor State (Bits 0 - 3)

This 4-bit field indicates the current state of this processor:

0000 - Initialization
0001 - Idle
0010 - Process Selection
0011 - Process Binding
0100 - Process Execution
0101 - Process Suspension
0110
thru - Reserved
1111

Dispatching Mode (Bits 4 - 5)

This 2-bit field determines which mode is to be used to dispatch this processor:

00 - Use normal port and carrier (Normal Mode)
01 - Use alarm port and carrier (Alarm Mode)
10 - Use reconfiguration port and carrier (Reconfiguration Mode)
11 - Use diagnostic port and carrier (Diagnostic Mode)

Stopped by IPC (Bit 6)

This bit indicates the stopped status of the processor, as determined by an Interprocessor Communication (IPC):

0 - Processor will execute a process if a process is available to execute.
1 - Processor is stopped by an IPC message and cannot execute a process until it receives a START IPC message.

Global IPC Accept Mode (Bit 7)

This bit determines whether Global interprocessor messages are currently being accepted/acknowledged:

0 - Global interprocessor messages are not being accepted nor acknowledged.
1 - Global interprocessor messages are being accepted and acknowledged.

Processor ID (Bits 8 - 15)

This 8-bit field is written by the associated processor at initialization time from externally read information. The value is read from interconnect address zero when the first local IPC signal is received by the processor after its Init pin is asserted.

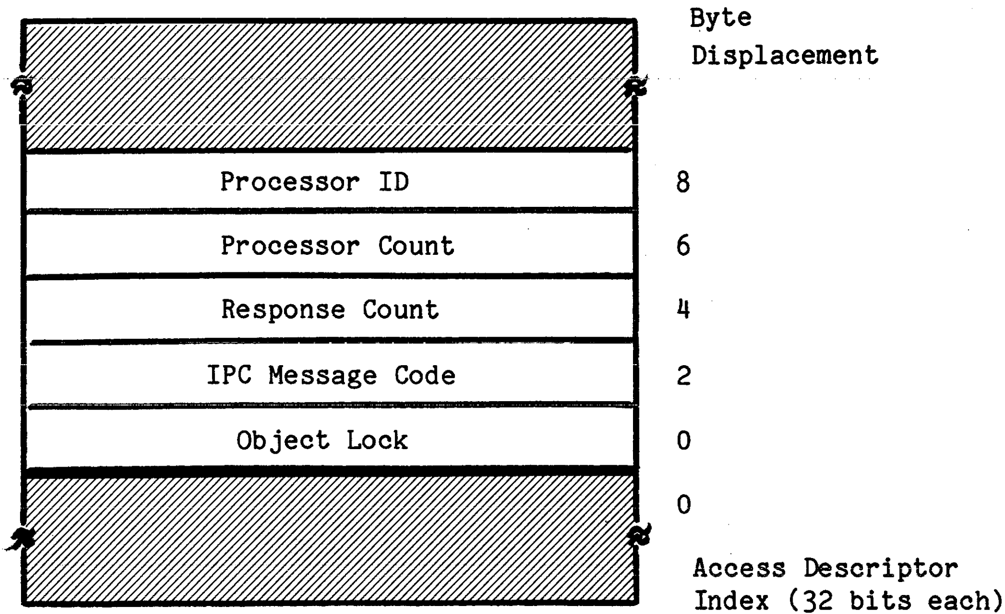
Processor Fault Data Area (Bytes 16 - 63)

This 48-byte data area is written by the processor after a processor-level fault and can be used by fault handling software. This area is described in the Fault and Trace Reference chapter of this manual.

TYPE RIGHTS FOR PROCESSOR OBJECTS

The type rights in an access descriptor that references a processor object are uninterpreted by the processor.

PROCESSOR COMMUNICATION OBJECT



Processor communication objects (PCOs) only require a data part that is interpreted by the processor. Typed refinements of PCOs are not supported by the GDP. The fields that constitute the processor-interpreted data part of a processor communication object are interpreted as follows:

Object Lock (Bytes 0 - 1)
This field provides mutually exclusive access to this processor communication object. The Object Lock field is defined for many system objects and is described in the first part of this chapter.

IPC Message Code (Bytes 2 - 3)
This 16-bit field contains one of the following function request encodings. Message codes 0 through 7 are IPC messages common between GDPs and Interface Processors (IPs). Message codes 8 through 14 are defined for GDPs but are ignored by IPs.

The following list gives the IPC Message Codes.

0 - Wakeup
1 - Start
2 - Stop
3 - Accept global IPCs
4 - Ignore global IPCs
5 - Requalify object table cache
6 - Reset processor
7 - Requalify processor
8 - Requalify process
9 - Requalify context
10 - Requalify data object cache
11 - Enter normal mode
12 - Enter alarm mode
13 - Enter reconfiguration mode
14 - Enter diagnostic mode

Response Count (Bytes 4 - 5)
This 16-bit field contains the number of processors remaining that have yet to respond to this IPC message. This field is initialized to the value in the Processor Count field (described below) during the execution of a SEND TO PROCESSOR instruction. As a processor receives the message, that processor acknowledges the IPC by decrementing this field.

Processor Count (Bytes 6 - 7)
The interpretation of this 16-bit field depends on whether this processor communication object is local or global. In a global PCO, this field contains the number of processors in global IPC accept mode using this PCO. In a local PCO, this field contains a fixed value of 1. This field should be updated by system software when processors are added or deleted in the system.

Processor ID (Bytes 8 - 9)
The interpretation of this 16-bit field depends on whether this PCO is local or global. In a global PCO, this field contains the fixed value of 0. In a local PCO, this field contains the processor ID of the associated processor in the low-order 8 bits with the high-order 8 bits being 0.

TYPE RIGHTS FOR PROCESSOR COMMUNICATION OBJECTS

The type rights in an access descriptor that references a processor communication object are interpreted as follows:

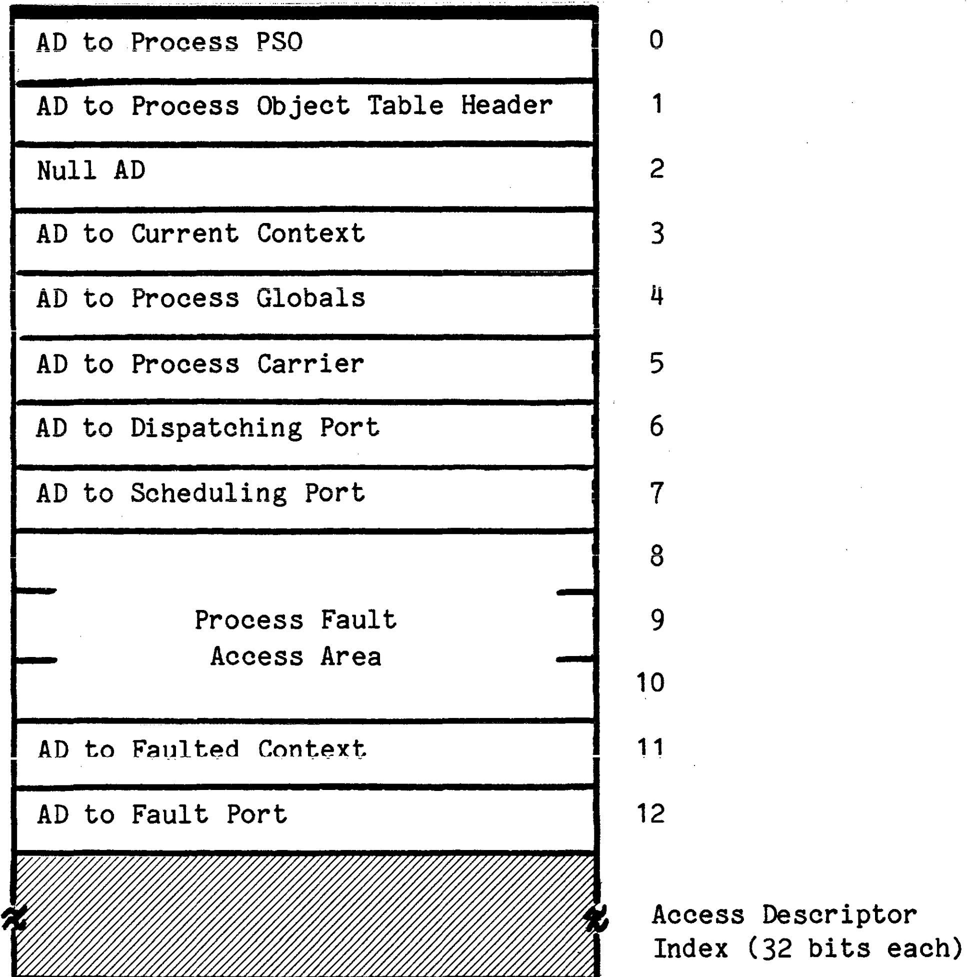
Type Right 1 — Send IPC Rights: If the bit is 1, an interprocessor message may be sent via this PCO.

Type Right 2 — Uninterpreted

Type Right 3 — Uninterpreted

PROCESS OBJECT

PROCESS OBJECT (ACCESS PART)



Refinement descriptors with process object as their object type are not supported by the GDP. The access descriptors that constitute the processor-interpreted portion of a process access part are interpreted as follows:

Process PSO (AD 0)
This AD references the associated physical storage object used for stack allocation of storage for this process.

Process Object Table Header (AD 1)
This AD references the object table header used for stack allocation of object descriptors for this process.

Null AD (AD 2)
This AD must be null and corresponds to the AD in an SRO that references a storage claim object. A null AD is interpreted as an infinite storage claim. Thus, no direct limit can be put on stack storage allocated for a process.

Current Context (AD 3)
This AD references the currently active context object of this process.

Process Globals (AD 4)
This AD references a global object indirectly accessible by all contexts within this process, using the COPY PROCESS GLOBALS operator. This object is defined by software.

Process Carrier (AD 5)
This AD references the process carrier associated with this process for use in interprocess communication and/or dispatching.

Dispatching Port (AD 6)
This AD references the dispatching port to which the carrier of this process is routed after expiration of the process's service period.

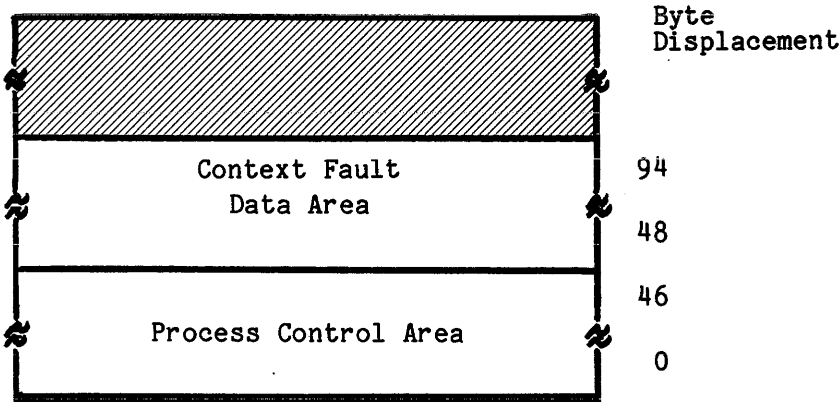
Scheduling Port (AD 7)
This AD references the port where the carrier of this process is routed when the period count of the process has expired; i.e., the process has used up all its time-slices.

Process Fault Access Area (ADs 8 - 10)
These 3 ADs are written by the processor after a process fault and can be used by fault handling software. They are described in the Fault and Trace Reference chapter of this manual.

Faulted Context (AD 11)
This AD references the faulted context object. This field is initially null and is defined only when written by the processor after a context fault (the Context Faulted bit in the process status is 1).

Fault Port (AD 12)
When a process fault occurs for this process, this AD is used as the reference to the port where the process carrier of this process is routed.

PROCESS OBJECT (DATA PART)



The areas that constitute the processor-interpreted portion of the data part of a process object are interpreted as follows:

Process Control Area (Bytes 0 - 47)
This 48-byte area serves as a control and status area for this process. It is described later in this chapter.

Context Fault Data Area (Bytes 48 - 95)
This 48-byte data area is written by the processor after a context fault and can be used by fault handling software. It is described in the Fault and Trace Reference chapter in this manual.

TYPE RIGHTS FOR PROCESS OBJECTS

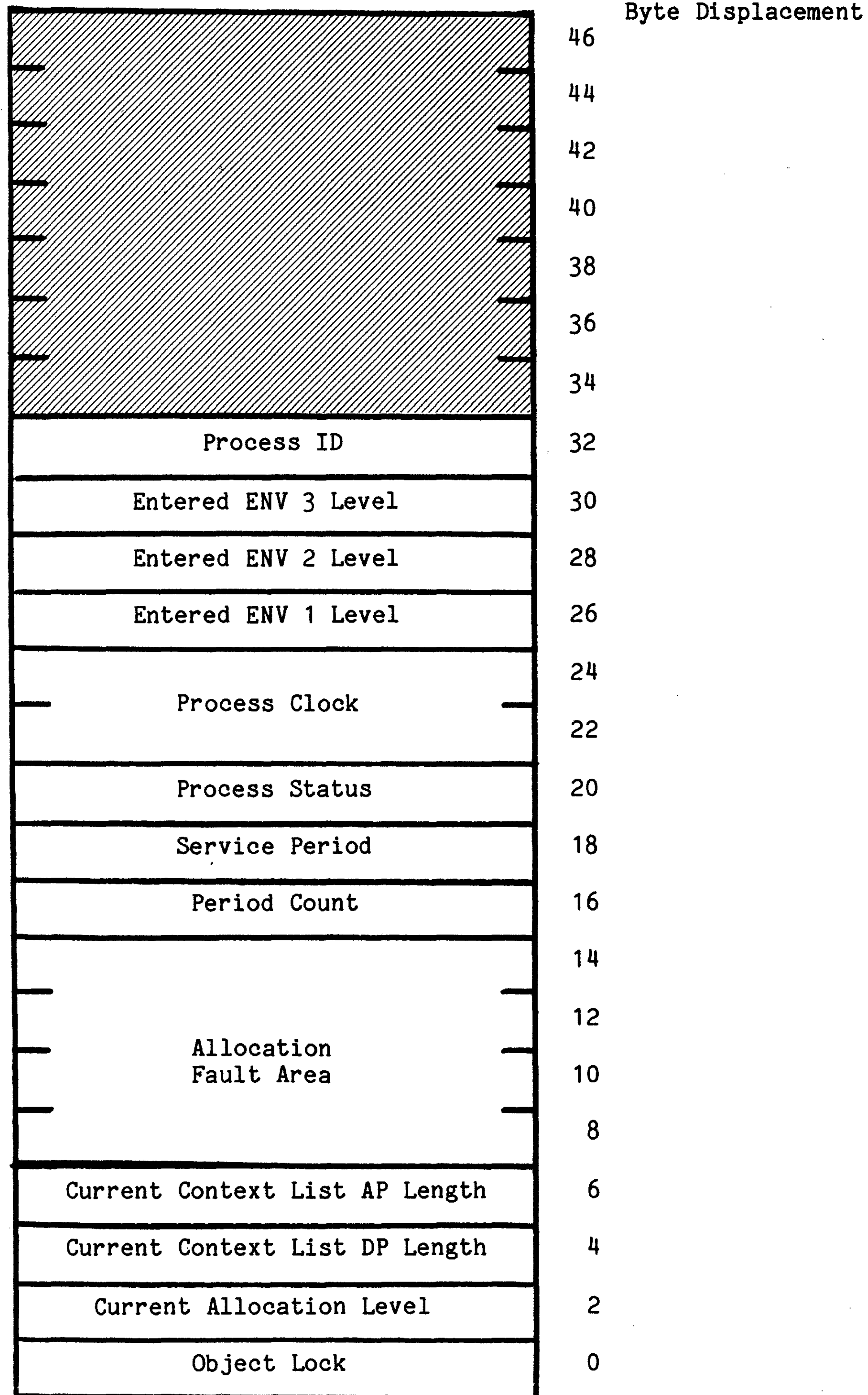
The type rights in an access descriptor that references a process object are interpreted as follows:

Type Right 1 — Set Mode Rights: If the bit is 1, the SET PROCESS MODE operator may be used to change the current process status.

Type Right 2 — Uninterpreted

Type Right 3 — Uninterpreted

PROCESS CONTROL AREA



The fields that constitute the processor-interpreted portion of the Process Control area are interpreted as follows:

Object Lock (Bytes 0 - 1)
This 16-bit field provides mutually exclusive access to the process object and its current context object. The Object Lock field is defined for many system objects and is described in the first part of this chapter.

Current Allocation Level (Bytes 2 - 3)
This 16-bit field is used when an object descriptor is allocated from the process allocation stack. This 16-bit value is used to initialize the Level field of the newly allocated object descriptor. The current allocation level is incremented by CALL or CALL THROUGH DOMAIN instructions and decremented by RETURN or RETURN AND FAULT instructions.

Current Context List DP Length (Bytes 4 - 5)
This 16-bit field contains the actual allocated length-1 of the data part of a context object in the pre-allocated context list.

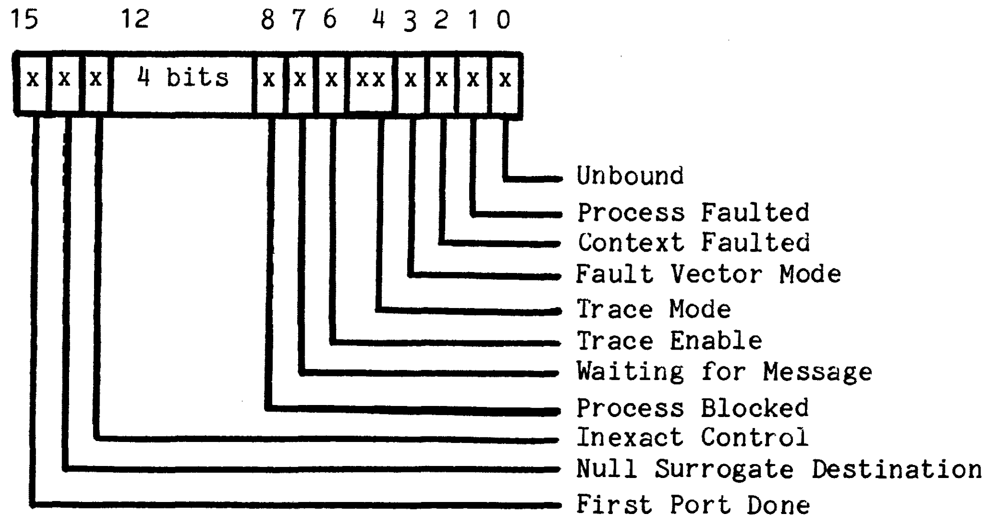
Current Context List AP Length (Bytes 6 - 7)
This 16-bit field contains the actual allocated length-1 of the access part of a context object in the pre-allocated context list.

Allocation Fault Area (Bytes 8 - 14)
This 8-byte area is written by the processor when an allocation-related fault occurs. This area can be used by fault handling software. It is described in the Fault and Trace Reference chapter in this manual.

Period Count (Bytes 16 - 17)
The value in this 16-bit field is one less than the number of service periods granted this process by a processor before the process is routed to its scheduling port. If this value is all 1s (16#FFFF#), the process is never sent to a scheduling port.

Service Period (Bytes 18 - 19)
The value in this 16-bit field is the maximum period (in system time units) over which a processor will serve this process before preemption.

Process Status (Bytes 20 - 21)
The format of the process status field is shown below:



The fields that constitute process status are interpreted as follows:

Unbound (Bit 0)
This bit indicates whether this process is unbound (i.e., not currently being executed by a processor):

0 - This process is bound to a processor.
1 - This process is not bound to a processor.

Process Faulted (Bit 1)
This bit indicates whether this process has faulted:

0 - Not faulted
1 - Faulted

Context Faulted (Bit 2)
This bit indicates whether the current active context has faulted:

0 - Not faulted
1 - Faulted

Fault Vector Mode (Bit 3)
This bit determines whether a process-level fault is to be treated as a process-level fault or a context-level fault:

0 - Treat process-level fault as context-level fault
1 - Treat process-level fault as process-level fault

Trace Mode (Bits 4 - 5)
This 2-bit field determines the current trace mode in effect for this process:

00 - No Trace Mode
01 - Fault Trace Mode
10 - Flow Trace Mode
11 - Full Trace Mode

Trace Enable (Bit 6)
This bit contains the trace rights bit from the current instruction object AD:

0 - Tracing is disabled
1 - Tracing is allowed

Waiting for Message (Bit 7)
This bit indicates whether this process is waiting for a message:

0 - This process is not blocked on a receive.
1 - Place the AD for the incoming message in the Interprocess Message location of the context access part when this process is resumed.

Process Blocked (Bit 8)
This bit indicates whether this process has been preempted before a possible trace is serviced:

0 - This process has been not pre-empted.
1 - This process has been preempted before a possible trace is serviced. If the trace mode in the process status is Full Trace, and the Trace Rights associated with the current instruction object AD is 1, a trace event needs to be generated before resumption of the process.

Inexact Control (Bit 13)
This bit determines whether to fault if a floating-point instruction produces an inexact result:

0 - No fault on inexact result
1 - Fault on inexact result

Null Surrogate Destination (Bit 14)
This bit indicates if the current surrogate operation has a destination.

0 - Valid surrogate destination
1 - Null surrogate destination

First Port Done (Bit 15)
This bit indicates whether the operation has completed at the first port of a two-port operation:

0 - First port operation has completed
1 - First port operation has not completed

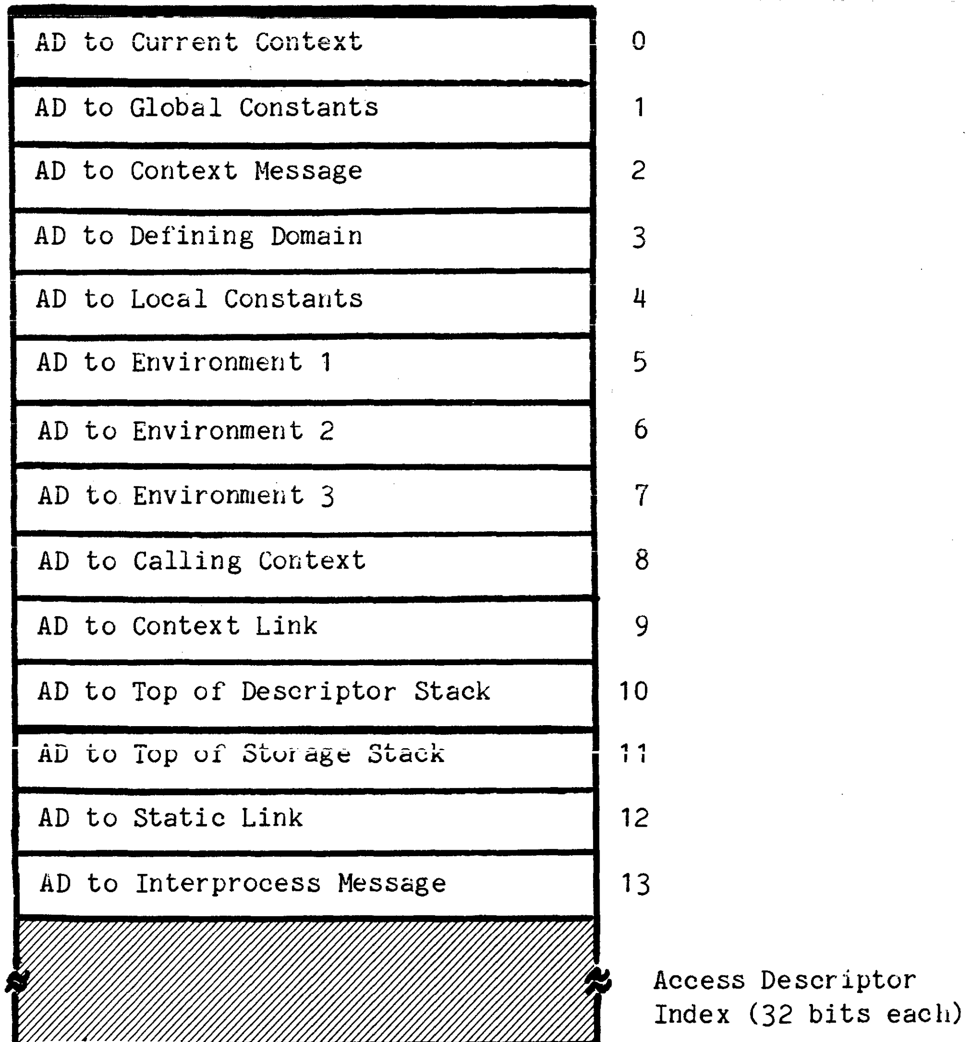
Process Clock (Bytes 22 - 25)
This ordinal value is the total processor execution time (in system time units) received by this process. This field is intialized to zero at process creation and is incremented by the system clock while this process is bound to a processor. The field may be used to accurately time a process independent of system scheduling activity.

Entered ENV 1 Level (Bytes 26 - 27)
Entered ENV 2 Level (Bytes 28 - 29)
Entered ENV 3 Level (Bytes 30 - 31)
These three 16-bit values in the process object contain level numbers for entered access environments 1 through 3. In each case, if the access environment is a refinement, then the level number is the level number of the root storage descriptor. If the corresponding access environment is null, then the level number is all 1s (0FFFFH).

Process ID (Bytes 32 - 33)
The low-order 2 bits of this 16-bit field must be 00. The high-order 14 bits contain the actual process ID (supplied by software).

CONTEXT OBJECT

CONTEXT OBJECT (ACCESS PART)



The access descriptors that constitute the processor-interpreted access part of a context object are interpreted as follows:

Current Context (AD 0)
This AD references this context itself.

Global Constants (AD 1)
This AD references an object containing frequently used data constants. All contexts reference the same Global Constants object which must be the same as that referenced by the Global Constants AD in the processor object.

Context Message (AD 2)
This AD references a refinement of the calling context object, which is used for parameter passing to the called context.

Defining Domain (AD 3)
This AD references the defining domain specified in the call instruction that was used to activate the current context.

Local Constants (AD 4)
This AD references an object containing local data constants associated with the current context.

Environment 1 (AD 5)
Environment 2 (AD 6)
Environment 3 (AD 7)
Each of these three ADs references an Environment object for the context. The Current Context access part and Environments 1 - 3 collectively constitute the instantaneous access environment of this context. When a context is called, the defining domain is entered as Environment 1 as part of the CALL operation. The ADs for Environments 2 - 3 are initially null and remain so until ADs are entered by using access environment manipulation operators.

Calling Context (AD 8)
This AD references the context object of the calling context and normally has Return rights.

Context Link (AD 9)
This AD references the next free context in the linked-list of pre-allocated contexts.

Top of Descriptor Stack (AD 10)
This AD references the most recent object descriptor allocated from the process allocation stack by this context or its calling contexts.

Top of Storage Stack (AD 11)
This AD references the most recent storage descriptor allocated from the process allocation stack by this context or its calling contexts.

Static Link (AD 12)
This AD is the static link AD passed with the CALL instruction. It is used to reference lexical level for supporting compiler implementation.

Interprocess Message (AD 13)
This AD is initially null. It references the most recent message received through a RECEIVE or successful CONDITIONAL RECEIVE instruction.

DELETE RIGHTS OF ADs IN THE CONTEXT OBJECT

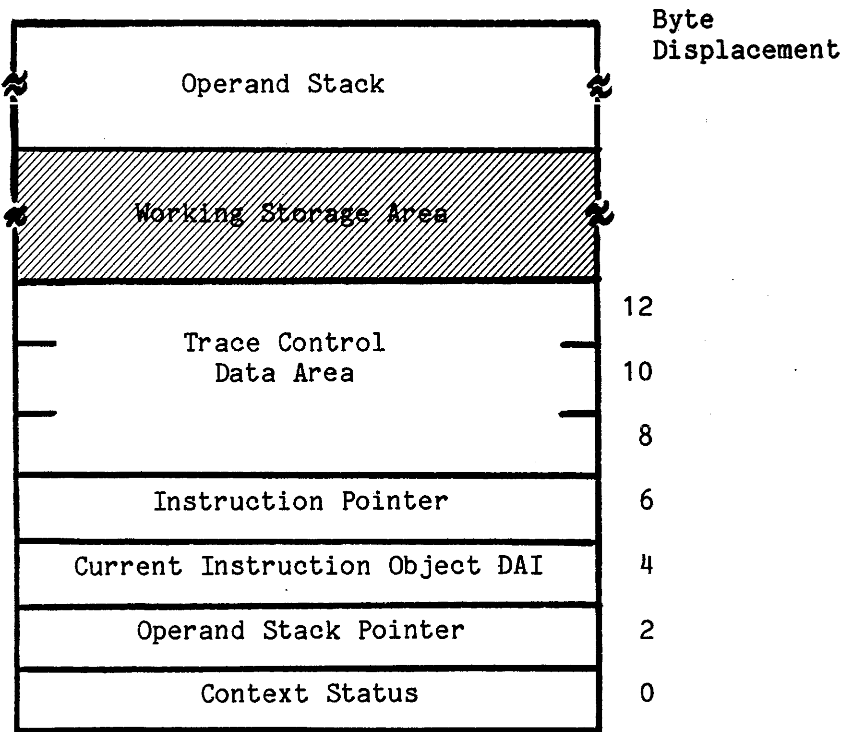
The following access descriptors in the context object are created without delete rights:

Current Context (AD 0)
Global Constants (AD 1)
Context Message (AD 2)
Defining Domain (AD 3)
Environment 1 - 3 (ADs 5 - 7)
Calling Context (AD 8)
Context Link (AD 9)
Top of Descriptor Stack (AD 10)
Top of Storage Stack (AD 11)

The following access descriptors in the context object are not changed from call to call:

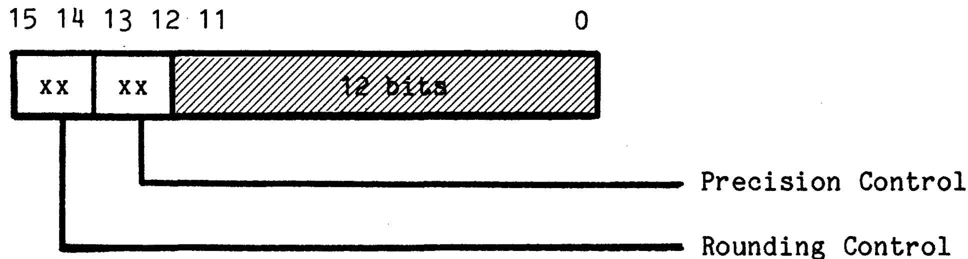
Current Context (AD 0)
Global Constants (AD 1)
Context Message (AD 2)
Calling Context (AD 8)
Context Link (AD 9)

CONTEXT OBJECT (DATA PART)



The fields that constitute the processor-interpreted data part of a context object are interpreted as follows:

Context Status (Bytes 0 - 1)
This 16-bit field contains status information pertinent to this context. The format of the context status field is shown below:



The fields that constitute context status are interpreted as follows:

Precision Control (Bits 12 - 13)
This field determines what precision is in effect for floating-point computation:

00 - Temporary-real precision
01 - Real precision
10 - Short-real precision
11 - Reserved

Rounding Control (Bits 14 - 15)
This field determines which rounding mode is in effect for floating-point computation:

00 - Round Nearest
01 - Round Up
10 - Round Down
11 - Round Toward Zero (truncate)

Operand Stack Pointer (Bytes 2 - 3)
This 16-bit field contains the byte displacement into the current context data part and points to the first free byte on the operand stack. This field is undefined when the context is currently active or faulted. The operand stack is 16 bits wide and, thus, the pointer is maintained with alignment to double-byte boundaries.

Current Instruction Object DAI (Bytes 4 - 5)
This 16 bit field contains the domain access index of the current instruction object. This field is undefined when the context is currently active or faulted.

Instruction Pointer (Bytes 6 - 7)
This 16-bit field contains the bit displacement from the base (fence) of the current instruction object to the next instruction to be executed. This field is undefined when the context is currently active or faulted.

Trace Control Data Area (Bytes 8 - 13)
This 6-byte area contains control information used in tracing. It is described in the Fault and Trace Reference chapter of this manual.

Working Storage Area
This area is preserved by the processor and may be used by software as working storage for this context. The initial value of the operand stack pointer specifies the beginning of the operand stack and thus the end of the Working Storage Area.

Operand Stack
This area of the context data part constitutes the operand stack for this context. This stack is 16 bits wide and grows upward (i.e., toward higher addresses in memory), limited only by the size of the context data part.

TYPE RIGHTS FOR CONTEXT OBJECTS

The type rights in an access descriptor that references a context object are interpreted as follows:

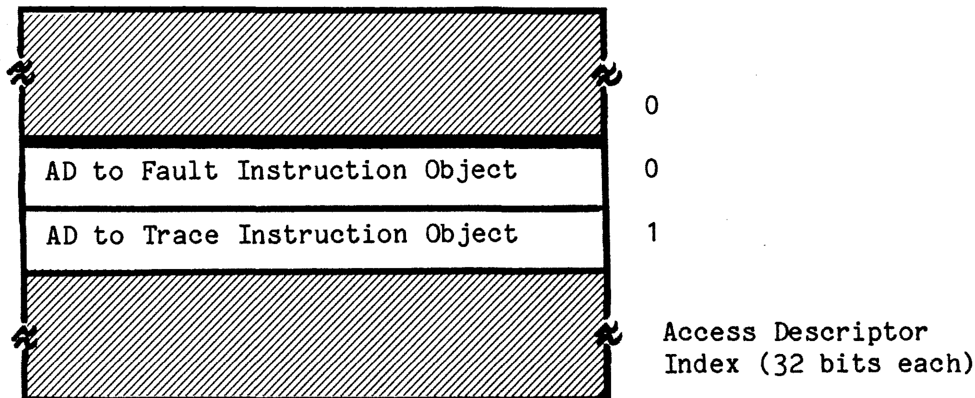
Type Right 1 Return Rights: If the bit is 1, the referenced context may be returned to.

Type Right 2 Uninterpreted

Type Right 3 Uninterpreted

DOMAIN OBJECT

The access part of a domain object is described below. Domain objects must have a data part, though the data part has no processor interpreted fields. Typed refinements of domain objects are supported by the GDP. However, the defining domain of a context cannot be a refinement. (A domain refinement can still be specified in a CALL THROUGH DOMAIN instruction; the instruction automatically traverses the refinement and writes an AD for the entire new defining domain into the called context.)



The access descriptors that constitute the processor-interpreted access part of a domain object are interpreted as follows:

Fault Instruction Object (AD 0)
: When a context-level fault occurs, control is transferred by a branch to bit displacement 64 in the instruction object referenced by this AD.

Trace Instruction Object (AD 1)
: When a trace event occurs, control is transferred by a branch to bit displacement 64 in the instruction object referenced by this AD.

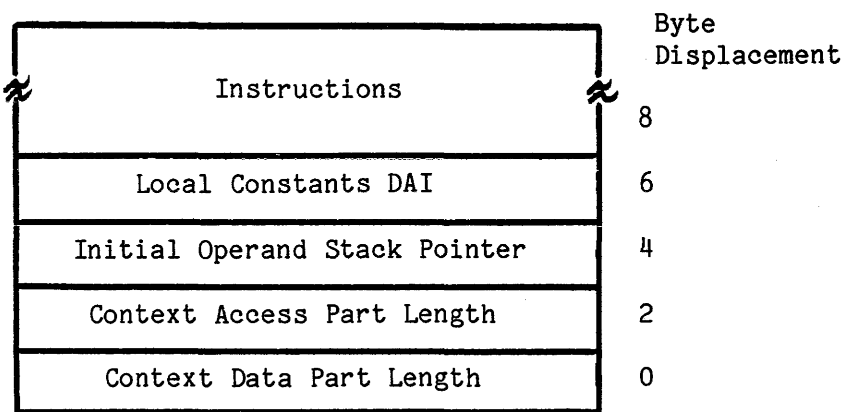
TYPE RIGHTS FOR DOMAIN OBJECTS

The type rights in an access descriptor that references a domain object are uninterpreted by the processor.

INSTRUCTION OBJECT

An instruction object is represented by a data part that is less than or equal to 8,192 bytes (65,536 bits) in length. An instruction object has no processor-interpreted access part. When created by software, the size of an instruction object should be rounded up to a 16-bit boundary plus 32 bits. This rounding is needed because the GDP fetches instructions in 32-bit units aligned on 16-bit boundaries. Displacement into an instruction object is always measured in bits, since instructions are variable bit-length and are not necessarily an integral number of bytes.

INSTRUCTION OBJECT (DATA PART)



The information in the first 4 double-byte fields of an instruction object is called the Instruction Object Header. It is only defined in instruction objects from which a context may be created on behalf of a Call instruction. Refinement descriptors with instruction object as their object type are not supported by the GDP. The fields that constitute an instruction object are interpreted as follows:

Context Data Part Length (Bytes 0 - 1)
: This 16-bit field contains a value that is one less than the length, in bytes, of the context data part. This value must be >= 15.

Context Access Part Length (Bytes 2 - 3)
: This 16-bit field contains a value that is one less than the length, in bytes, of the context access part. This value must be >= 63.

Initial Operand Stack Pointer (Bytes 4 - 5)
: This 16-bit field contains a byte displacement into the specific context data part associated with the invocation of this instruction object. It is initialized to be the displacement to the first byte of the operand stack. This value must be even.

Local Constants DAI (Bytes 6 - 7)

This 16-bit field contains the domain access index of the AD for the current object from which data constant operands are accessed. This DAI specifies the AD to be copied into the Local Constants location (AD 4) in the context object that is associated with the invocation of this instruction object.

Instructions

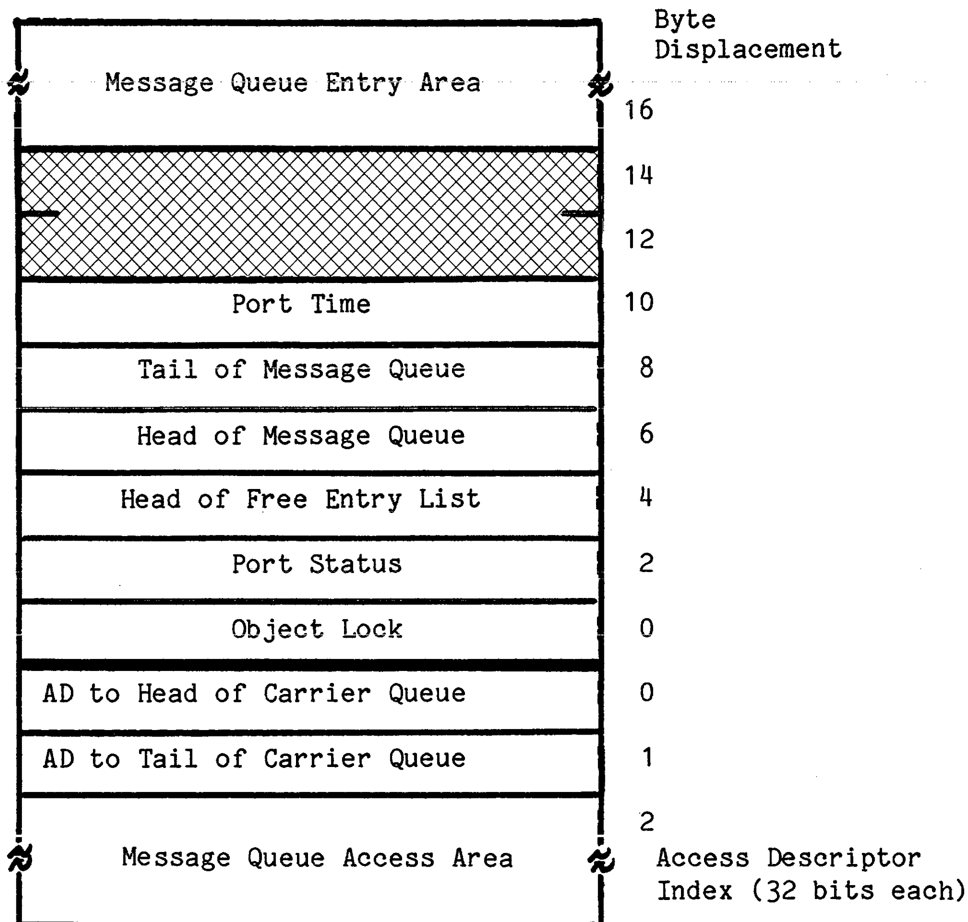
The remaining area of the instruction object is for instructions. The first instruction must start at bit displacement 64.

TYPE RIGHTS FOR INSTRUCTION OBJECTS

The type rights in an access descriptor that references an instruction object are interpreted as follows:

Type Right 1	Uninterpreted
Type Right 2	Trace Rights: If the bit is 1, the instruction object can be traced.
Type Right 3	Uninterpreted

PORT OBJECT



Refinement descriptors with port object as their object type are not supported by the GDP. The access descriptors that constitute the processor-interpreted access part of a port object are interpreted as follows:

Head of Carrier Queue (AD 0)
: This AD references the carrier object at the head of the carrier queue. If the queue is empty, this AD is null.

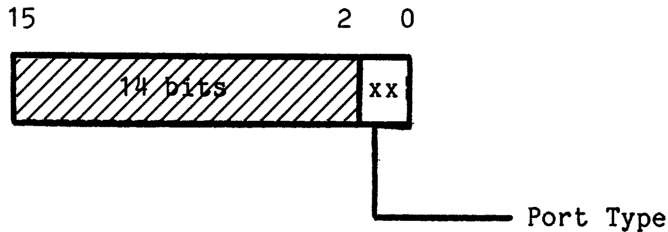
Tail of Carrier Queue (AD 1).
: This AD references the carrier object at the tail of the carrier queue. If the queue is empty, this AD is null.

Message Queue Access Area
: This is a fixed-length area for access descriptors to message objects. There is a one-to-one correspondence between these ADs and the port message queue entries in this port's data part (described below). If the associated queue entry is a free entry, its corresponding AD in this access area is null.

The fields that constitute the processor-interpreted data part of a port object are interpreted as follows:

Object Lock (Bytes 0 - 1)
This 16-bit object lock provides mutually exclusive access to the port object and all associated blocked carriers. The Object Lock field is defined for many system objects and is described in the first part of this chapter.

Port Status (Bytes 2 - 3)
The format of the 16-bit port status field is shown below:



The fields that constitute port status are interpreted as follows:

Port Type (Bits 0 - 1)
This field indicates the port type and determines the message queuing and dequeuing policy in effect at the port:

00 - FIFO (First In First Out)
01 - Priority
10 - Deadline within Priority
11 - Delay

Head of Free Entry List (Bytes 4 - 5)
This 16-bit field contains the byte displacement into the port data part of the head of a linked list of free message queue entries. If there are no free queue entries, this value is zero.

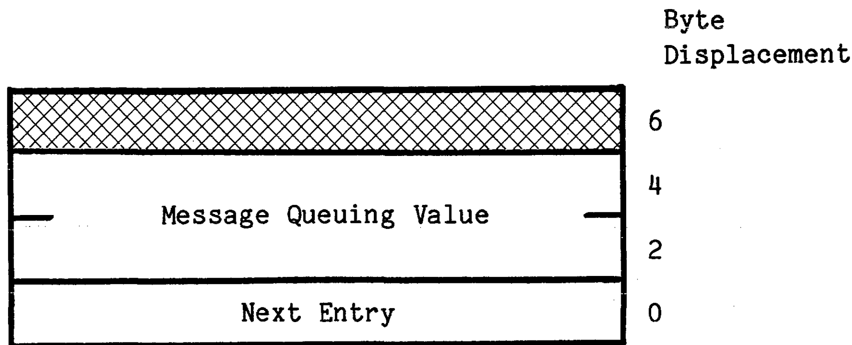
Head of Message Queue (Bytes 6 - 7)
This 16-bit field contains the byte displacement into the port data part of the first entry in the port message queue.

Tail of Message Queue (Bytes 8 - 9)
This 16-bit field contains the byte displacement into the port data part of the last entry in the port message queue.

Port Time (Bytes 10 - 11)
This 16-bit field contains the processor clock value at the time the port deadline information was last updated in the message queue.

Message Queue Entry Area (Starting at Byte 16)
This is a fixed-length array of message queue entries containing both a linked list of free entries and a linked list of port message queue entries. For each message queue entry (i.e., for both message and free entries) there is a corresponding access descriptor in the message queue access area of this port object. Both free entries and message queue entries are represented by the same message queue entry format shown below:

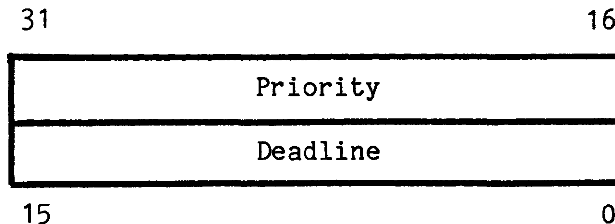
MESSAGE QUEUE ENTRY



The fields that constitute a message queue entry are interpreted as follows:

Next Entry (Bytes 0 - 1)
This 16-bit field contains the byte displacement into the port data part to the next queue entry (either free entry or message queue entry).

Message Queuing Value (Bytes 2 - 5)
This 32-bit field contains priority and deadline values for this message queue entry. The message queuing value is not interpreted for FIFO ports. A message queuing value has the following format:



For priority and deadline ports, the fields that constitute a message queuing value are interpreted as follows:

Deadline (Bits 0 - 15)
This 16-bit field contains a 2's complement value (in the range of -2**14 to 2**14-1) representing the relative deadline of this message queue entry with respect to the previous message queue entry.

Priority (Bits 16 - 31)
This 16-bit value determines the priority order of message entries in the queue--low values are low priority. Entries with the same priority are ordered at insertion by their deadline.

TYPE RIGHTS FOR PORT OBJECTS

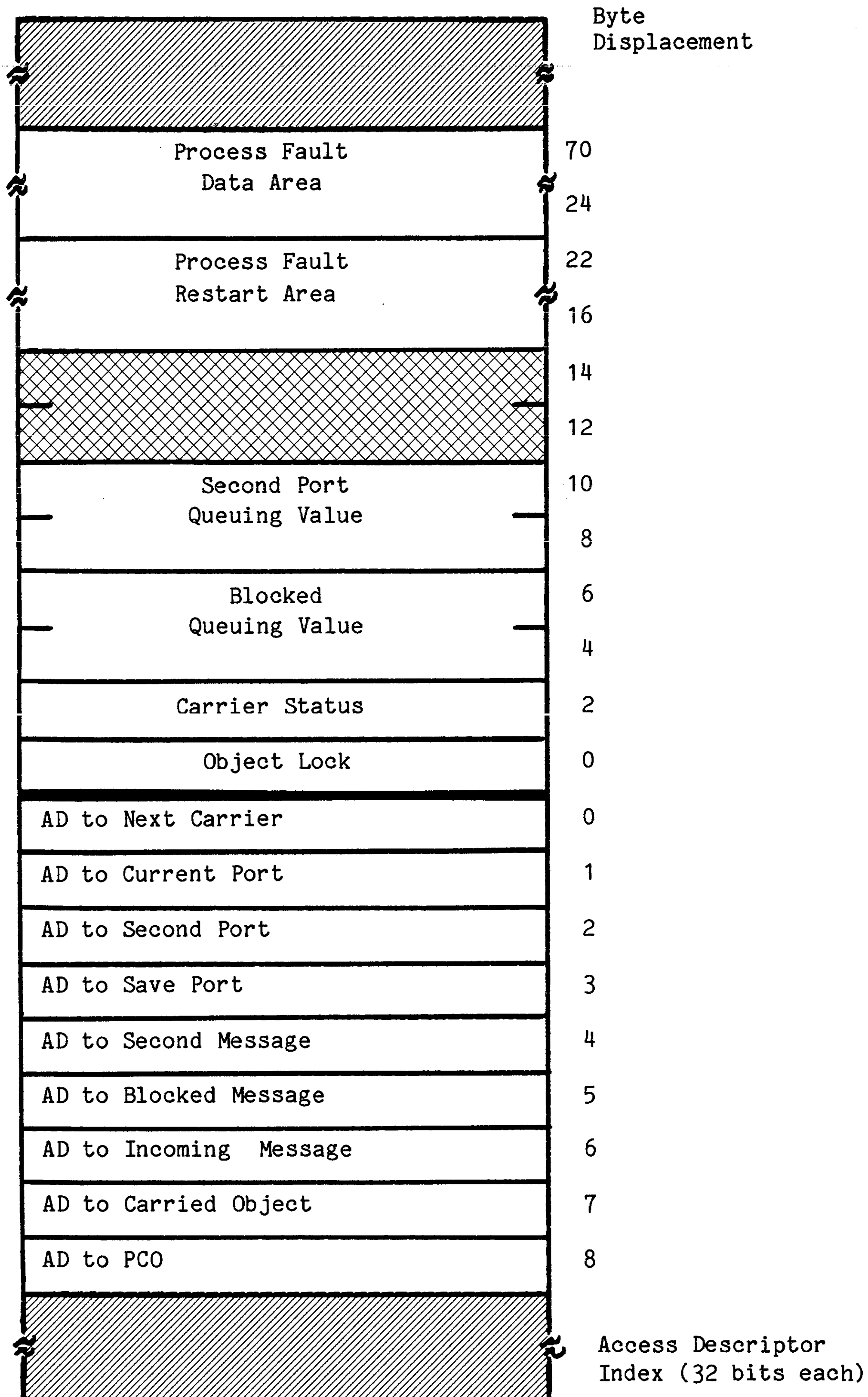
The type rights in an access descriptor that references a port object are interpreted as follows:

Type Right 1 — Send Rights: If the bit is 1, a message may be sent using this port.

Type Right 2 — Receive Rights: If the bit is 1, a message may be received using this port.

Type Right 3 — Send Process Rights: If the bit is 1, a process may be forwarded using this port.

CARRIER OBJECT



The access descriptors that constitute the processor-interpreted access part of a carrier object are interpreted as follows:

Next Carrier (AD 0)
    This AD references the next carrier in the carrier queue. This AD is null if the carrier is not in a carrier queue.

Current Port (AD 1)
    This AD references the port at which the carrier is enqueued. This AD is null if the carrier is not queued at a port.

Second Port (AD 2)
    This AD references the second port to which this carrier is forwarded.

Save Port (AD 3)
    This AD is interpreted only in process carriers waiting at the delay port. When the process carrier is removed from the delay port, it is forwarded to the port referenced by this AD.

Second Message (AD 4)
    This AD references the message used in forwarding this carrier. In processor and process carriers, this is an AD for the carrier itself. In surrogate carriers, this AD references a refinement of the carrier beginning at the incoming message AD location (described below).

Blocked Message (AD 5)
    If this carrier is enqueued as the result of a blocked SEND, this AD references the message being sent.

Incoming Message (AD 6)
    If a message is received, this AD references the message received.

Carried Object (AD 7)
    For processor and process carriers, this AD references the corresponding processor or process object.

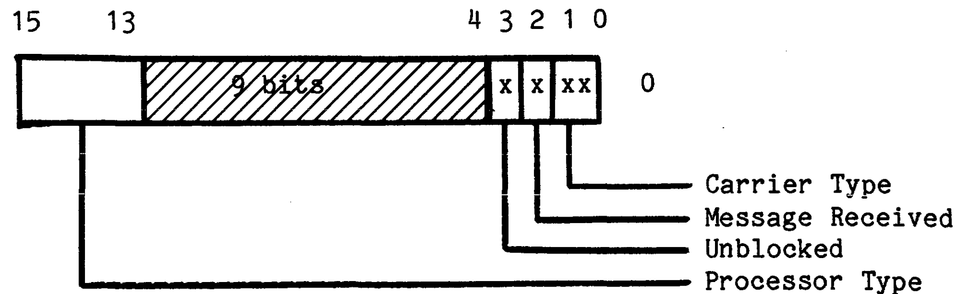
PCO (AD 8)
    For processor carriers, this AD references the Processor Communication Object associated with the carried object.

Note that ADs 7 and 8 are not interpreted by the processor in surrogate carriers.

The fields that constitute the processor-interpreted data part of a carrier object are interpreted as follows:

Object Lock (Bytes 0 - 1)
This 16-bit field provides mutually exclusive access to the carrier object. The Object Lock field is defined for many system objects and is described in the first part of this chapter.

Carrier Status (Bytes 2 - 3)
The format of the carrier status field is shown below:



The fields that constitute carrier status are interpreted as follows:

Carrier Type (Bits 0 - 1)
This 2-bit field indicates the type of this carrier:

00 - Processor Carrier.
01 - Process Carrier.
10 - Surrogate Carrier.
11 - Reserved

Message Received (Bit 2)
This bit indicates whether the Incoming Message AD location contains an AD to a received message:

0 - Carrier object has not received message.
1 - Carrier object has received message.

Unblocked (Bit 3)
This bit indicates whether this carrier is blocked to receive a message:

0 - Carrier is blocked to receive a message.
1 - Carrier is not blocked to receive a message.

Processor Type (Bits 13-15)
This field is only interpreted for processor carriers, in which case it contains the processor type (GDP or IP, never All) of the associated processor.

Blocked Queuing Value (Bytes 4 - 7)
If the carrier is blocked on a SEND, this 32-bit queuing value is used on its behalf to complete the SEND operation when the carrier becomes unblocked. This value is updated whenever the carrier is blocked. The fields in a queuing value are described earlier in this chapter as part of the port object description.

Second Port Queuing Value (Bytes 8 - 11)
If the carrier is a surrogate or forwarded carrier, this 32-bit queuing value is used to complete the SURROGATE SEND or forwarding operation. This value is supplied by software. The fields in a queuing value are described earlier in this chapter as part of the port object description.

Process Fault Restart Area (Bytes 16 - 23)
This 8-byte data area is interpreted by the processor during process binding to allow restarting a faulted process without changing the Operand Stack Pointer, Current Instruction Object DAI, and Instruction Pointer in the current context. It is described in more detail in the Fault and Trace Reference chapter of this manual. This field is interpreted only in Process Carriers. It is reserved in all other types of carriers.

Process Fault Data Area (Bytes 24 - 70)
This 48-byte data area is written by the processor after a process level fault and can be used by fault handling software. It is described in the Fault and Trace Reference chapter in this manual. This field is interpreted only in Process Carriers. It is reserved in all other types of carriers.

TYPE RIGHTS FOR CARRIER OBJECTS

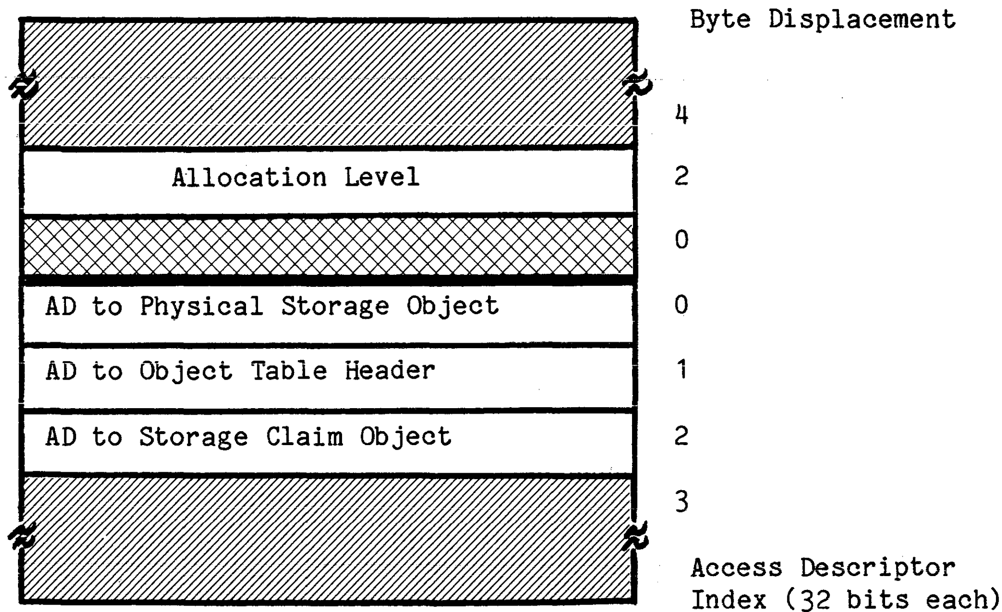
The type rights in an access descriptor that references a carrier object are interpreted as follows:

Type Right 1 — Surrogate Rights: If the bit is 1, the carrier may be used with surrogate instructions.

Type Right 2 — Uninterpreted

Type Right 3 — Uninterpreted

STORAGE RESOURCE OBJECT



Refinement descriptors with storage resource object as their object type are not supported by the GDP. The access descriptors that constitute the processor-interpreted access part of a Storage Resource Object (SRO) are interpreted as follows:

Physical Storage Object (AD 0)
: This AD references the associated physical storage object used for heap allocation of storage associated with this SRO.

Object Table Header (AD 1)
: This AD references the object table header used for heap allocation of object descriptors associated with this SRO.

Storage Claim Object (AD 2)
: This AD references the associated storage claim object used for heap allocation from this SRO. A null AD is interpreted as an infinite storage claim.

The fields that constitute the processor-interpreted data part of a storage resource object are interpreted as follows:

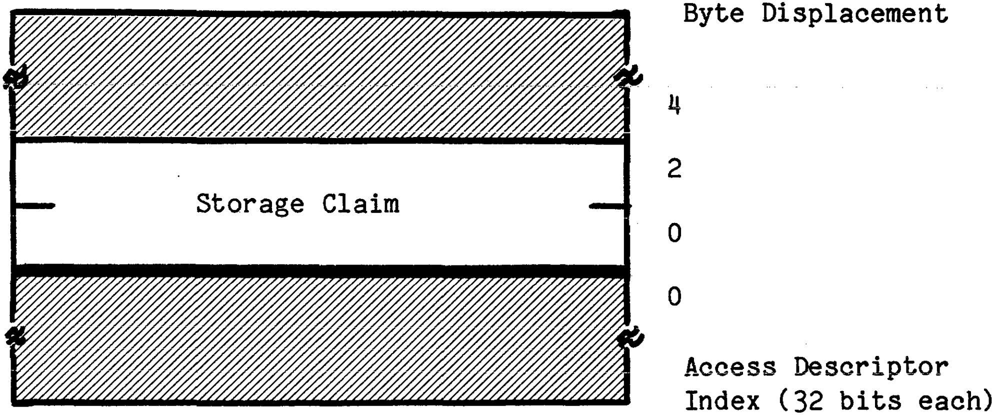
Allocation Level (Bytes 2 - 3)
: This 16-bit field is used when an object descriptor is allocated from the object table associated with the SRO. The 16-bit Allocation Level is used to initialize the level field of the newly allocated object descriptor.

TYPE RIGHTS FOR STORAGE RESOURCE OBJECTS

The type rights in an access descriptor that references a storage resource object are interpreted as follows:

Type Right 1	Create Rights: If the bit is 1, physical storage and/or object descriptor space may be allocated for object creation from the SRO.
Type Right 2	Uninterpreted
Type Right 3	Uninterpreted

STORAGE CLAIM OBJECT



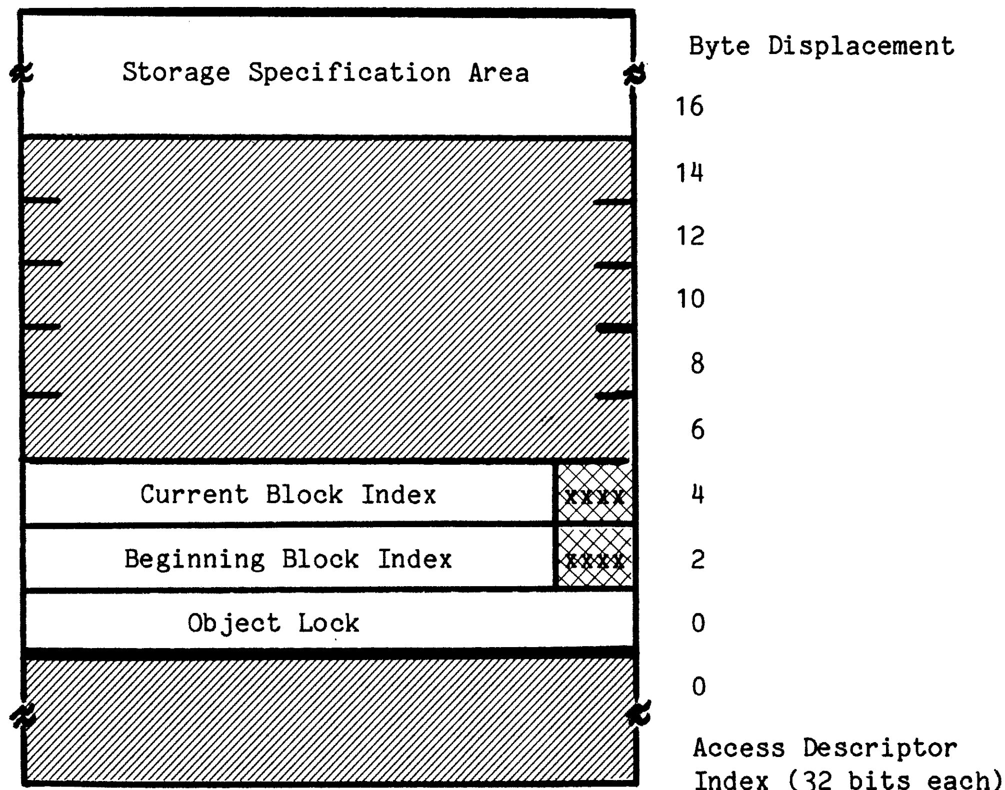
Storage Claim Objects (SCOs) consist of only a data part that is interpreted by the processor. The fields that constitute the data part of a storage claim object are interpreted as follows:

Storage Claim (Bytes 0 - 3)
This 32-bit integer indicates the number of bytes of storage (including each allocated segment's prefix and pad) that can be allocated using an SRO. The amount of allocated storage is indivisibly subtracted from the storage claim by the processor when storage is allocated from the PSO associated with the SRO. When storage allocated using this claim object is deallocated, system software should add back the amount of deallocated storage to the storage claim.

TYPE RIGHTS FOR STORAGE CLAIM OBJECTS

The type rights in an access descriptor that references a storage claim object are uninterpreted by the processor.

PHYSICAL STORAGE OBJECT



Physical Storage Objects (PSOs) consist of only a data part that is interpreted by the processor. The fields that constitute the processor-interpreted data part of a physical storage object are interpreted as follows:

Object Lock (Bytes 0 - 1)
This 16-bit object lock provides mutually exclusive access to this physical storage object. The object lock field is defined for many system objects and is described in the first part of this chapter.

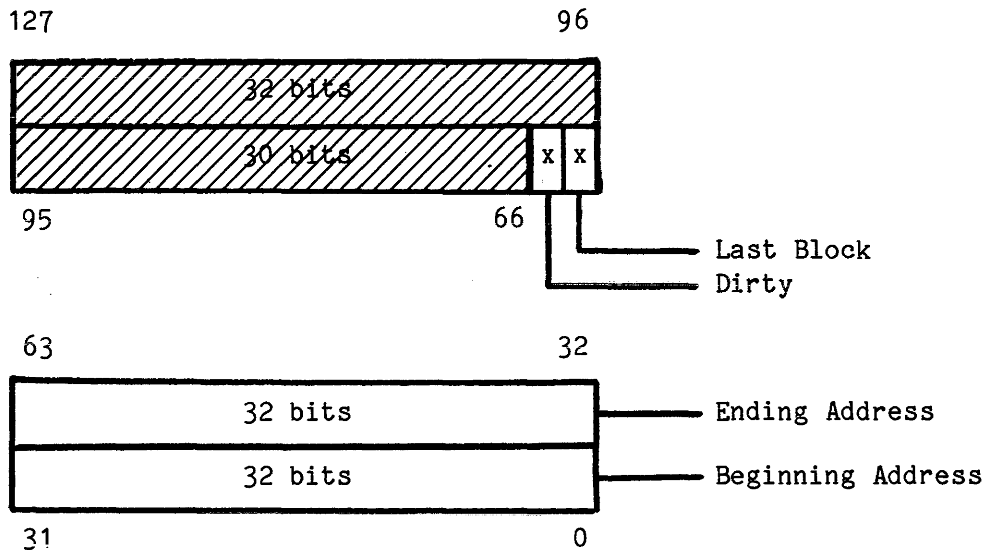
Beginning Block Index (Bytes 2 - 3)
This 12-bit field contains an index into this physical storage data part to the first storage block specifier located in the Storage Specification Area. The value in this field counts 16-byte storage block specifiers and must be >= 1.

Current Block Index (Bytes 4 - 5)
This 12-bit field contains an index into this physical storage data part to the storage block specifier at which the rotating first-fit search starts for the next storage allocation cycle. The value in this field counts 16-byte storage block specifiers and must be >= 1.

Storage Specification Area (Beginning at Byte 16)

This area contains an array of 16-byte storage block specifiers used for allocation from this physical storage object. The array can vary in size. There can only be one storage block specifier active at a time in a PSO that is used for the current process allocation stack. This single storage block specifier is indexed by the same value in both the Beginning Block Index and Current Block Index fields of this PSO. A storage block specifier represents an available block of storage and has the following format:

STORAGE BLOCK SPECIFIER



The fields that constitute a storage block specifier are interpreted as follows:

Beginning Address (Bits 0 - 31)

This 32-bit field contains the physical address (in bytes in the storage address space) for the first byte of the block of available storage defined by this storage block specifier. The value must be <= 2**24 and must be an integral multiple of 8 (i.e., the least-significant 3 bits must be zero).

Ending Address (Bits 32 - 63)

This 32-bit field contains the physical address of one byte past the last available byte in the physical storage block defined by this storage block specifier. The value must be <= 2**24 and must be an integral multiple of 8 (i.e., the least-significant 3 bits must be zero).

Last Block (Bit 64)
This bit indicates whether this storage block specifier is the last one in this physical storage object:

0 - Not last storage block specifier
1 - Last storage block specifier

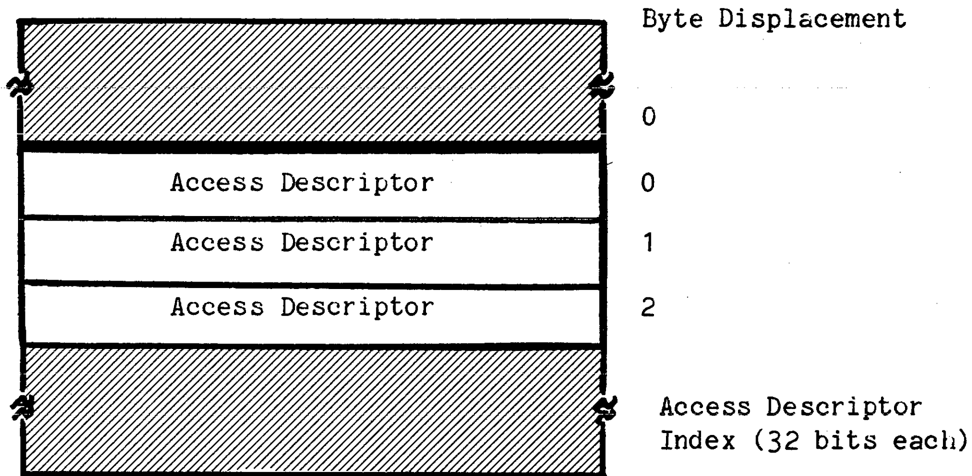
Dirty (Bit 65)
This bit indicates whether memory allocated using this storage block specifier needs to be initialized to zero:

0 - No initialization to zero (block already zeroed)
1 - Initialize to zero

TYPE RIGHTS FOR PHYSICAL STORAGE OBJECTS

The type rights in an access descriptor that references a physical storage object are uninterpreted by the processor.

TYPE DEFINITION OBJECT

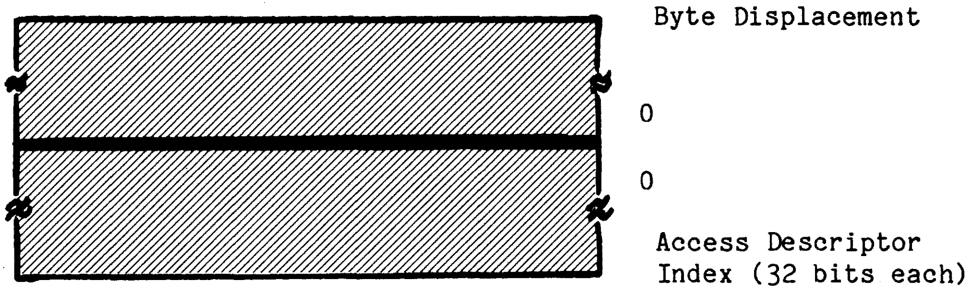


Type Definition Objects (TDOs) contain access descriptors for domains and other objects used to manage object instances of the dynamic type defined by this TDO.

TYPE RIGHTS FOR TYPE DEFINITION OBJECTS

The type rights in an access descriptor that references a type definition object are uninterpreted by the processor.

DYNAMIC TYPE OBJECT



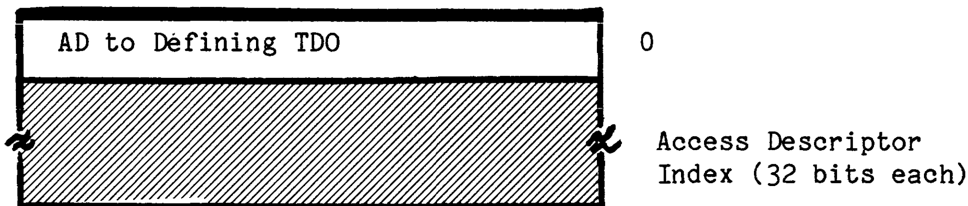
Dynamic Type Objects (DTOs) represent an instance of the Dynamic Type defined by the Defining TDO. They contain no predefined, processor-interpreted fields.

TYPE RIGHTS FOR DYNAMIC TYPE OBJECTS

The type rights in an access descriptor that references a dynamic type object are uninterpreted by the processor.

TYPE CONTROL OBJECT

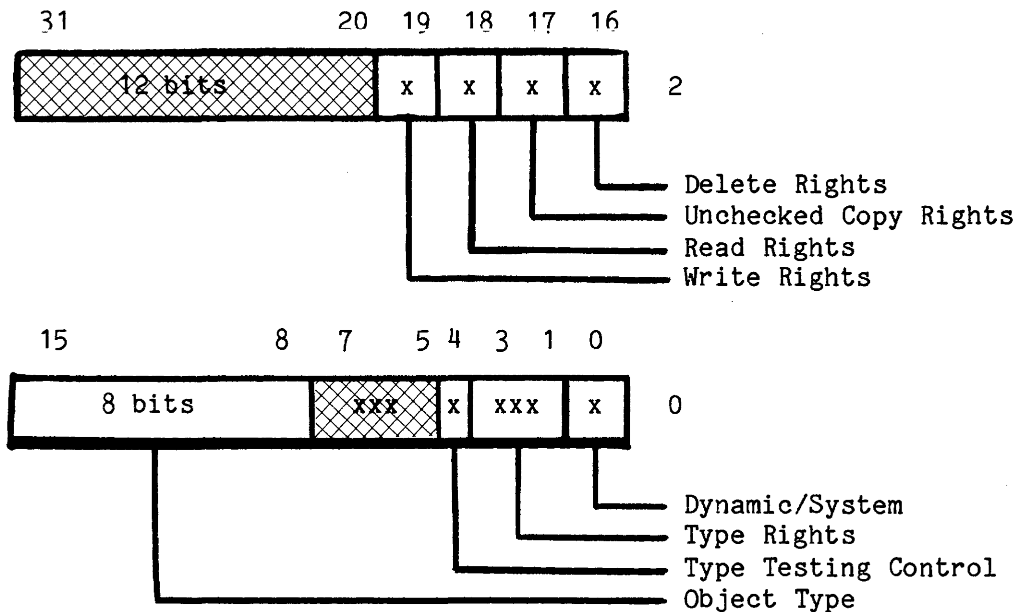
TYPE CONTROL OBJECT (ACCESS PART)



Refinement descriptors with type control object as their object type are not supported by the GDP. The access descriptors that constitute the processor-interpreted access part of a Type Control Object (TCO) are interpreted as follows:

Defining TDO (AD 0)
If the Dynamic/System bit is 0, this AD must be null. If the Dynamic/System bit is 1, this AD references a defining type definition object for the dynamic type.

TYPE CONTROL OBJECT (DATA PART)



Different TCO fields are used by the AMPLIFY RIGHTS, CREATE TYPED OBJECT, and CREATE TYPED REFINEMENT operators. The fields that constitute the processor-interpreted data part of a type control object are interpreted as follows:

Dynamic/System (Bit 0)
If this bit is 1, the TCO is for a user dynamic type object. Otherwise, the TCO is for a system object.

Type Rights (Bits 1 - 3)
Delete Rights (Bit 16)
Unchecked Copy Rights (Bit 17)
Read Rights (Bit 18)
Write Rights (Bit 19)
During rights amplification of an AD, these values are logically ORed to their corresponding rights fields in the access descriptor to be amplified.

Type Testing Control (Bit 4)
This bit is interpreted during an AMPLIFY RIGHTS instruction as follows:

0 - This TCO can be used to amplify any access descriptor.
1 - This TCO can only be used to amplify access descriptors for objects whose object type matches that specified in the Object Type field of this TCO.

Object Type (Bits 8 - 15)
This 8-bit field is interpreted the same as the corresponding field in object descriptors. In typed object creation, the Object Type field in this TCO is written to the corresponding field in the new OD for the object being created. In amplification, these fields are used in type testing control (see the Type Testing Control field described above). In refinement creation, this field defines the object type of the source object from which the refinement is obtained and also defines the object type of the newly created refinement.

TYPE RIGHTS FOR TYPE CONTROL OBJECTS

The type rights in an access descriptor that references a type control object are interpreted as follows:

Type Right 1 — Create Rights: If the bit is 1, the TCO may be used in typed object creation.

Type Right 2 — Amplify Rights: If the bit is 1, the TCO may be used in rights amplification.

Type Right 3 — Refine Rights: If the bit is 1, the TCO may be used in typed refinement creation.

This chapter defines the operator set of the iAPX 432 General Data Processor. Each GDP operator description specifies the operator encoding and describes the operands required in an instruction using the operator. The algorithmic action of each operator is also described.

FUNCTIONAL INDEX OF OPERATORS

In the following functional index of operators, the GDP operators are functionally grouped in operator Identification Number order. The operator ID# precedes each operator name. Some functional groupings begin with a set of sub-operator procedures that are used later in the actual operator descriptions of that group. Operators marked with an asterisk (*) are identical to operators with the same assigned ID# and are specified by the same Class and Opcode fields. Operators marked with a double asterisk (**) do not have a unique operator ID#. They are classified as absolute branches and relative branches. Absolute branches have an operator ID# of 254, while relative branches have an operator ID# of 255.

DATA OPERATORS

Character Operators	Mnemonic	Page
1 Move Character	MOV_C	10-15
2 Zero Character	ZRO_C	10-15
3 One Character	ONE_C	10-15
4 Save Character	SAV_C	10-15
5 AND Character	AND_C	10-16
6 Inclusive OR Character	IOR_C	10-16
7 Exclusive OR Character	XOR_C	10-16
8 Equivalence Character	EQV_C	10-16
9 NOT Character	NOT_C	10-17
10 Add Character	ADD_C	10-17
11 Subtract Character	SUB_C	10-17
12 Increment Character	INC_C	10-17
13 Decrement Character	DEC_C	10-17

14 Equal Character .... EQL_C 10-18
15 Not Equal Character .... NEQ_C 10-18
16 Equal Zero Character .... EQZ_C 10-18
17 Not Equal Zero Character .... NEZ_C 10-18
18 Less Than Character .... LSS_C 10-19
19 Less Than or Equal Character .... LEQ_C 10-19

20 Convert Character to Short Ordinal ...... CVT_C_SO 10-19
21 Convert Character to Integer .... CVT_C_I 10-19

Short-Ordinal Operators

* 22 Move Short Ordinal .... MOV_SO 10-20
* 23 Zero Short Ordinal .... ZRO_SO 10-20
* 24 One Short Ordinal .... ONE_SO 10-20
* 25 Save Short Ordinal .... SAV_SO 10-20

26 AND Short Ordinal .... AND_SO 10-21
27 Inclusive OR Short Ordinal .... IOR_SO 10-21
28 Exclusive OR Short Ordinal .... XOR_SO 10-21
29 Equivalence Short Ordinal .... EQV_SO 10-21
30 NOT Short Ordinal .... NOT_SO 10-22

31 Extract Short Ordinal .... EXT_SO 10-22
32 Insert Short Ordinal .... INS_SO 10-22
33 Significant Bit Short Ordinal .... SIG_SO 10-22

34 Add Short Ordinal .... ADD_SO 10-23
35 Subtract Short Ordinal .... SUB_SO 10-23
36 Increment Short Ordinal .... INC_SO 10-23
37 Decrement Short Ordinal .... DEC_SO 10-23
38 Multiply Short Ordinal .... MUL_SO 10-23
39 Divide Short Ordinal .... DIV_SO 10-24
40 Remainder Short Ordinal .... REM_SO 10-24

* 41 Equal Short Ordinal .... EQL_SO 10-24
* 42 Not Equal Short Ordinal .... NEQ_SO 10-24
* 43 Equal Zero Short Ordinal .... EQZ_SO 10-24
* 44 Not Equal Zero Short Ordinal .... NEZ_SO 10-25
45 Less Than Short Ordinal .... LSS_SO 10-25
46 Less Than or Equal Short Ordinal .... LEQ_SO 10-25

47 Convert Short Ordinal to Integer .... CVT_SO_I 10-25

Short-Integer Operators

	No.	Operator	Mnemonic	Page
*	22	Move Short Integer	MOV_SI	10-26
*	23	Zero Short Integer	ZRO_SI	10-26
*	24	One Short Integer	ONE_SI	10-26
*	25	Save Short Integer	SAV_SI	10-26
	48	Add Short Integer	ADD_SI	10-27
	49	Subtract Short Integer	SUB_SI	10-27
	50	Increment Short Integer	INC_SI	10-27
	51	Decrement Short Integer	DEC_SI	10-27
	52	Negate Short Integer	NEG_SI	10-27
	53	Multiply Short Integer	MUL_SI	10-28
	54	Divide Short Integer	DIV_SI	10-28
	55	Remainder Short Integer	REM_SI	10-28
*	41	Equal Short Integer	EQL_SI	10-28
*	42	Not Equal Short Integer	NEQ_SI	10-29
*	43	Equal Zero Short Integer	EQZ_SI	10-29
*	44	Not Equal Zero Short Integer	NEZ_SI	10-29
	56	Less Than Short Integer	LSS_SI	10-29
	57	Less Than or Equal Short Integer	LEQ_SI	10-30
	58	Positive Short Integer	PTV_SI	10-30
	59	Negative Short Integer	NTV_SI	10-30
	60	Move in Range Short Integer	MIR_SI	10-30
	61	Convert Short Integer to Integer	CVT_SI_I	10-31

Ordinal Operators

	No.	Operator	Mnemonic	Page
*	62	Move Ordinal	MOV_O	10-32
*	63	Zero Ordinal	ZRO_O	10-32
*	64	One Ordinal	ONE_O	10-32
*	65	Save Ordinal	SAV_O	10-32
	66	AND Ordinal	AND_O	10-33
	67	Inclusive OR Ordinal	IOR_O	10-33
	68	Exclusive OR Ordinal	XOR_O	10-33
	69	Equivalence Ordinal	EQV_O	10-33
	70	NOT Ordinal	NOT_O	10-34
	71	Extract Ordinal	EXT_O	10-34
	72	Insert Ordinal	INS_O	10-34
	73	Significant Bit Ordinal	SIG_O	10-34
	74	Add Ordinal	ADD_O	10-34
	75	Subtract Ordinal	SUB_O	10-35
	76	Increment Ordinal	INC_O	10-35
	77	Decrement Ordinal	DEC_O	10-35
	78	Multiply Ordinal	MUL_O	10-35
	79	Divide Ordinal	DIV_O	10-35
	80	Remainder Ordinal	REM_O	10-36
	81	Index Ordinal	IDX_O	10-36

	No.	Operator	Mnemonic	Page
*	82	Equal Ordinal	EQL_O	10-37
*	83	Not Equal Ordinal	NEQ_O	10-37
*	84	Equal Zero Ordinal	EQZ_O	10-37
*	85	Not Equal Zero Ordinal	NEZ_O	10-37
	86	Less Than Ordinal	LSS_O	10-38
	87	Less Than or Equal Ordinal	LEQ_O	10-38
*	88	Convert Ordinal to Integer	CVT_O_I	10-38
	89	Convert Ordinal to Temporary Real	CVT_O_TR	10-38

Integer Operators

	No.	Operator	Mnemonic	Page
*	62	Move Integer	MOV_I	10-39
*	63	Zero Integer	ZRO_I	10-39
*	64	One Integer	ONE_I	10-39
*	65	Save Integer	SAV_I	10-39
	90	Add Integer	ADD_I	10-40
	91	Subtract Integer	SUB_I	10-40
	92	Increment Integer	INC_I	10-40
	93	Decrement Integer	DEC_I	10-40
	94	Negate Integer	NEG_I	10-40
	95	Multiply Integer	MUL_I	10-41
	96	Divide Integer	DIV_I	10-41
	97	Remainder Integer	REM_I	10-41
*	82	Equal Integer	EQL_I	10-41
*	83	Not Equal Integer	NEQ_I	10-42
*	84	Equal Zero Integer	EQZ_I	10-42
*	85	Not Equal Zero Integer	NEZ_I	10-42
	98	Less Than Integer	LSS_I	10-42
	99	Less Than or Equal Integer	LEQ_I	10-43
	100	Positive Integer	PTV_I	10-43
	101	Negative Integer	NTV_I	10-43
	102	Move in Range Integer	MIR_I	10-43
	103	Convert Integer to Character	CVT_I_C	10-44
	104	Convert Integer to Short Ordinal	CVT_I_SO	10-44
	105	Convert Integer to Short Integer	CVT_I_SI	10-44
*	88	Convert Integer to Ordinal	CVT_I_O	10-44
	106	Convert Integer to Temporary Real	CVT_I_TR	10-44

Short-Real Operators

	No.	Operator	Mnemonic	Page
*	62	Move Short Real	MOV_SR	10-45
*	63	Zero Short Real	ZRO_SR	10-45
*	65	Save Short Real	SAV_SR	10-45
	107	Add Short Real	ADD_SR	10-46
	108	Add Temporary Real to Short Real	ADD_TR_SR	10-46
	109	Add Short Real to Temporary Real	ADD_SR_TR	10-46
	110	Subtract Short Real	SUB_SR	10-46
	111	Subtract Temporary Real from Short Real	SUB_TR_SR	10-47
	112	Subtract Short Real from Temporary Real	SUB_SR_TR	10-47
	113	Multiply Short Real	MUL_SR	10-47
	114	Multiply Temporary Real by Short Real	MUL_TR_SR	10-47
	115	Multiply Short Real by Temporary Real	MUL_SR_TR	10-48
	116	Divide Short Real	DIV_SR	10-48
	117	Divide Temporary Real into Short Real	DIV_TR_SR	10-48
	118	Divide Short Real into Temporary Real	DIV_SR_TR	10-48
	119	Negate Short Real	NEG_SR	10-49
	120	Absolute Value Short Real	ABS_SR	10-49
	121	Equal Short Real	EQL_SR	10-49
	122	Equal Zero Short Real	EQZ_SR	10-49
	123	Less Than Short Real	LSS_SR	10-49
	124	Less Than or Equal Short Real	LEQ_SR	10-50
	125	Positive Short Real	PTV_SR	10-50
	126	Negative Short Real	NTV_SR	10-50
	127	Convert Short Real to Temporary Real	CVT_SR_TR	10-50

Real Operators

No.	Operator	Mnemonic	Page
128	Move Real	MOV_R	10-51
129	Zero Real	ZRO_R	10-51
130	Save Real	SAV_R	10-51
131	Add Real	ADD_R	10-52
132	Add Temporary Real to Real	ADD_TR_R	10-52
133	Add Real to Temporary Real	ADD_R_TR	10-52
134	Subtract Real	SUB_R	10-52
135	Subtract Temporary Real from Real	SUB_TR_R	10-53
136	Subtract Real from Temporary Real	SUB_R_TR	10-53
137	Multiply Real	MUL_R	10-53
138	Multiply Temporary Real by Real	MUL_TR_R	10-53
139	Multiply Real by Temporary Real	MUL_R_TR	10-54
140	Divide Real	DIV_R	10-54
141	Divide Temporary Real into Real	DIV_TR_R	10-54
142	Divide Real into Temporary Real	DIV_R_TR	10-54
143	Negate Real	NEG_R	10-55
144	Absolute Value Real	ABS_R	10-55

145 Equal Real .... EQL_R 10-55
146 Equal Zero Real .... EQZ_R 10-55
147 Less Than Real .... LSS_R 10-55
148 Less Than or Equal Real .... LEQ_R 10-56
149 Positive Real .... PTV_R 10-56
150 Negative Real .... NTV_R 10-56

151 Convert Real to Temporary Real .... CVT_R_TR 10-56

Temporary-Real Operators

152 Move Temporary Real .... MOV_TR 10-57
153 Zero Temporary Real .... ZRO_TR 10-57
154 Save Temporary Real .... SAV_TR 10-57

155 Add Temporary Real .... ADD_TR 10-58
156 Subtract Temporary Real .... SUB_TR 10-58
157 Multiply Temporary Real .... MUL_TR 10-58
158 Divide Temporary Real .... DIV_TR 10-58
159 Remainder Temporary Real .... REM_TR 10-59
160 Negate Temporary Real .... NEG_TR 10-59
161 Square Root Temporary Real .... SQT_TR 10-59
162 Absolute Value Temporary Real .... ABS_TR 10-59

163 Equal Temporary Real .... EQL_TR 10-60
164 Equal Zero Temporary Real .... EQZ_TR 10-60
165 Less Than Temporary Real .... LSS_TR 10-60
166 Less Than or Equal Temporary Real .... LEQ_TR 10-60
167 Positive Temporary Real .... PTV_TR 10-61
168 Negative Temporary Real .... NTV_TR 10-61

169 Convert Temporary Real to Ordinal .... CVT_TR_O 10-61
170 Convert Temporary Real to Integer .... CVT_TR_I 10-61
171 Convert Temporary Real to Short Real .... CVT_TR_SR ... 10-62
172 Convert Temporary Real to Real .... CVT_TR_R 10-62

OBJECT OPERATORS

Sub-Operator Procedures

Set Copied 10-63
Level Check 10-63
Store AD 10-63
Object Locking 10-63
OD Allocation 10-64
Segment Allocation 10-64

Branch Operators

	No.	Operator	Mnemonic	Page
**		Branch	BR	10-65
**		Branch True	BR_T	10-65
**		Branch False	BR_F	10-65
	173	Branch Indirect	BR_INDIRECT	10-66
	174	Branch Intersegment	BR_ISEG	10-66
	175	Branch Intersegment without Trace	BR_ISEG_WO_TRACE	10-66
	176	Branch Intersegment and Link	BR_ISEG_LINK	10-67
	177	Breakpoint	BREAKPOINT	10-67

Access Descriptor Operators

	No.	Operator	Mnemonic	Page
	178	Copy Access Descriptor	COPY_AD	10-68
	179	Null Access Descriptor	NULL_AD	10-68

Type and Rights Operators

	No.	Operator	Mnemonic	Page
	180	Amplify Rights	AMPLIFY_RIGHTS	10-69
	181	Restrict Rights	RESTRICT_RIGHTS	10-70
	182	Retrieve Type Definition	RETRIEVE_TYP_DEF	10-71

Refinement Operators

	No.	Operator	Mnemonic	Page
	183	Create Refinement	CREATE_RFN	10-72
	184	Create Typed Refinement	CREATE_TYPED_RFN	10-74

Object Creation Operators

	No.	Operator	Mnemonic	Page
	185	Create Object	CREATE_OBJ	10-76
	186	Create Typed Object	CREATE_TYPED_OBJ	10-77

Access Inspection Operators

	No.	Operator	Mnemonic	Page
*	187	Inspect Access Descriptor	INSPECT_AD	10-78
	188	Inspect Object	INSPECT_OBJ	10-78
	189	Equal Access	EQL_ACCESS	10-79
	190	Move to Embedded Data Value	MOV_TO_EDV	10-79
*	187	Move from Embedded Data Value	MOV_FM_EDV	10-79

Access Interlock Operators

	No.	Operator	Mnemonic	Page
	191	Lock Object	LOCK_OBJ	10-80
	192	Unlock Object	UNLOCK_OBJ	10-80
	193	Indivisibly Add Short Ordinal	INDIV_ADD_SO	10-81
	194	Indivisibly Add Ordinal	INDIV_ADD_O	10-81
	195	Indivisibly Insert Short Ordinal	INDIV_INS_SO	10-82
	196	Indivisibly Insert Ordinal	INDIV_INS_O	10-82

Context Operators

Sub-Operator Procedures:
ENV Entry 10-83
Context Call 10-83

Operators:
197 Enter Environment 1 .... ENTER_ENV_1 10-85
198 Enter Environment 2 .... ENTER_ENV_2 10-85
199 Enter Environment 3 .... ENTER_ENV_3 10-85
200 Copy Process Globals .... COPY_PRCS_GLOBALS . 10-86
201 Set Context Mode .... SET_CTXT_MODE 10-86
202 Adjust Stack Pointer .... ADJ_SP 10-86
203 Call .... CALL 10-87
204 Call Through Domain................. CALL_THRU_DOMAIN .. 10-87
205 Return .... RET 10-88
206 Return and Fault .... RET_FAULT 10-88

Process Communication Operators

Sub-Operator Procedures:
Enqueue Message 10-89
Dequeue Message 10-89
Enqueue Carrier 10-90
Dequeue Carrier 10-90
Forward Carrier 10-90
Surrogate Common 10-90
Send Common 10-91
Receive Common 10-92

Operators:
207 Send .... SEND 10-93
208 Receive .... RECEIVE 10-94
209 Conditional Send .... COND_SEND 10-95
210 Conditional Receive .... COND_RECEIVE 10-96
211 Surrogate Send .... SUR_SEND 10-97
212 Surrogate Receive .... SUR_RECEIVE 10-98
213 Delay Process .... DELAY_PRCS 10-99
214 Send Process .... SEND_PRCS 10-100
215 Set Process Mode .... SET_PRCS_MODE 10-100
216 Read Process Clock .... READ_PRCS_CLOCK ... 10-101

Processor Communication Operators

217 Send to Processor .... SEND_PSOR 10-102
218 Read Processor Status .... READ_PSOR_STATUS .. 10-103

Interconnect Operators

219 Move to Interconnect .... MOV_TO_ICT 10-104
220 Move from Interconnect .... MOV_FM_ICT 10-104

Block Move Operator

222 Block Move .... BLK_MOV 10-105

OPERATOR DESCRIPTIONS

This section contains the operator descriptions for the GDP operator set. Each description includes the number and type of required operands and a commentary on the operation performed. The descriptions do not include full details on faulting. See the Fault and Trace Reference chapter of this manual for the descriptions of faults for each operator.

Each operator is described using a table of the following form:

OPERATOR NAME OPERATOR_MNEMONIC

ID#	Operands 1	2	3

Opcode	Reference	Format	Class
	varies	varies	

The table is split into two parts. The first part characterizes the operator by describing the order and type of operands it requires. The second part of the table shows the binary format of an instruction that encodes the operator. The binary values given for the Class and Opcode fields together uniquely determine a given operator. The Format and Reference fields are included for the sake of completeness--their actual binary encodings vary depending on the operand locations and the kind of operand addressing intended.

The labels for the columns in these tables have the following meaning:

ID#

This decimal number is an identifier number for the operator. Though it is not encoded in instructions, this number is used to identify each GDP operator. For example, it is generated by the processor to indicate (in fault data areas) the operator that was executing when a fault occurred.

Operands

The term operand has a precise meaning in this manual. See the Instruction Interface chapter of this manual for more details. Up to three operands are required for each operator. Operands are either explicitly or implicitly specified in an instruction and are addressed through the operand addressing mechanism of the GDP. Operands always reside in the data parts of objects. For a given instruction, the operand(s) are never encoded into the instruction stream as literals.

They are explicitly specified by a Data Reference field or implicitly specified by a particular Format field encoding. An implicit reference always specifies the current context's operand stack for the corresponding operand. In a particular instruction, the mapping of data references to operands is defined by the Format field in that instruction. A table of Format field encodings is included in the Instruction Encoding chapter of this manual.

Abbreviations are used in the operator tables to indicate operand types. These operand types are briefly described later in this chapter. The following abbreviations are used:

as	Access Selector (16 bits)
b	Boolean (8 bits)
bfs	Bit-Field Specifier (16 bits)
c	Character (8 bits)
dai	Domain Access Index (16 bits)
i	Integer (32 bits)
o	Ordinal (32 bits)
pd	Packed Double-word (64 bits)
pw	Packed Word (32 bits)
r	Real (64 bits)
si	Short Integer (16 bits)
so	Short Ordinal (16 bits)
sr	Short Real (32 bits)
tr	Temporary Real (80 bits)

Class

This field in an instruction consists of a variable-length bit string that is the class encoding for the given operator. All digits shown are significant and constitute the actual bit field encoded in the instruction for that operator.

Format

This field in an instruction consists of a variable-length bit string that is the format encoding. The specific encoding of the field varies depending on the order of the specific operator and the intended mapping of operand references in a given instruction. A table of format field encodings is included in the Instruction Composition chapter of this manual.

Reference

This field in an instruction contains 0 to 3 data references for the operands of the instruction. It can also contain 0 or 1 data references followed by a branch reference. The specific encoding of the field varies depending on the intended operand addressing modes for each operand. A description of reference field formats and encodings is included in the Instruction Composition chapter of this manual.

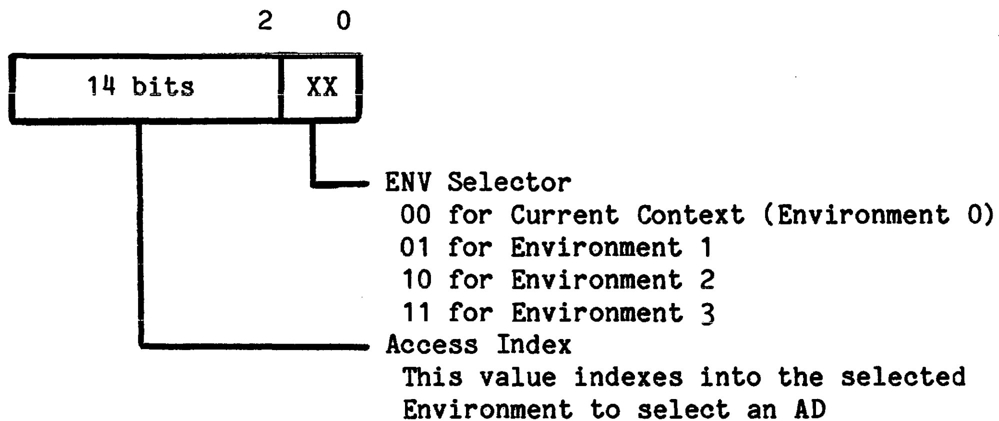
Opcode

This field in an instruction consists of a variable-length bit string that is the opcode encoding for the given operator. All digits shown are significant and constitute the actual bit field encoded in the instruction for that operator.

OPERAND TYPES

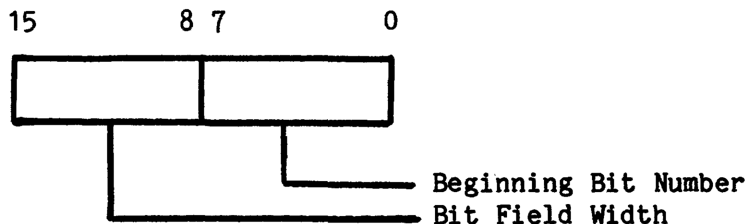
The types of operands that can be referenced in GDP instructions are briefly described in this section. For further details about the interpretation of those operand types that are also data types, see the Computational Data Types chapter of this manual.

as ACCESS SELECTOR (16 bits)
Access selectors select an AD in the access environment of the current context. Access selector operands have the following format:



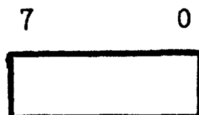
b BOOLEAN (8 bits)
A boolean is a value of type character that is used to represent logical TRUE or FALSE. TRUE is represented by xxxxxxx1 and FALSE by xxxxxxx0 (with the x bits being uninterpreted don't care bits).

bfs BIT-FIELD SPECIFIER (16 bits)
A bit-field specifier specifies a field of bits to be manipulated within an ordinal or short-ordinal operand. A bit-field specifier consists of two adjacent bytes as shown below:

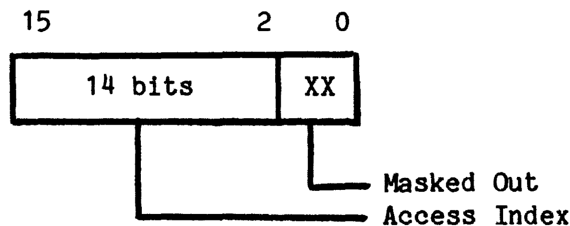


For short-ordinal bit-manipulation operators, only the low-order 4 bits of these bytes are interpreted by the GDP during execution. For ordinal bit-manipulation operators, only the low-order 5 bits of these bytes are interpreted by the GDP during execution. The first byte specifies the beginning bit of the field. The second (next higher-addressed) byte of a bfs specifies one less than the number of bits in the field. Thus a bit field within a short ordinal can contain from 1 to 16 bits; within an ordinal, from 1 to 32 bits. Bit fields that extend past the most significant bit of the short ordinal or ordinal containing them "wrap around." For example, a 4-bit field beginning at bit 14 within a short ordinal contains bits 14, 15, 0, and 1 (LSB to MSB).

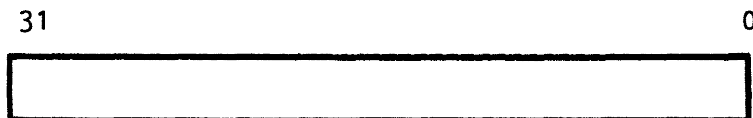
c CHARACTER (8 bits)
These 8-bit operands are used to represent booleans, text characters, or unsigned integers in the range 0 to 255.



dai DOMAIN ACCESS INDEX (16 bits)
A domain access index selects an access descriptor in the defining domain of the current context. Only the upper 14 bits are used for the AD index. The lower 2 bits are masked out. DAIs have the following format:

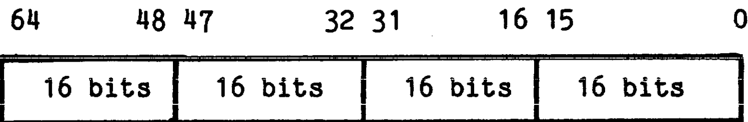


i INTEGER (32 bits)
These 32-bit operands represent signed integer values in the range -2,147,483,648 to 2,147,483,647 in 2's complement form.

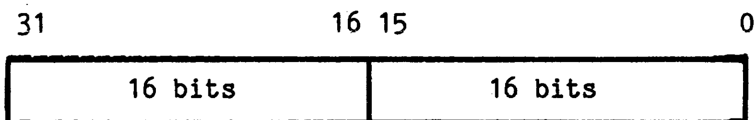


o ORDINAL (32 bits)
These 32-bit operands represent unsigned integer values in the range 0 to 4,294,967,295, or bit strings of 32 bits or less.

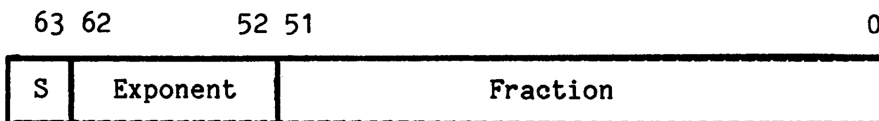
pd PACKED DOUBLEWORD (64 bits)
Packed doublewords consist of four 16-bit parts. Each part can contain one of the other 16-bit operand types. Packed doubleword operands are required by some object operators.



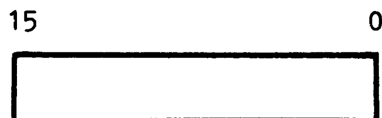
pw PACKED WORD (32 bits)
Packed words consist of two 16-bit parts. Each part can contain one of the other 16-bit operand types. Packed word operands are required by some object operators.



r REAL (64 bits)
These 64-bit operands represent floating-point numbers. See the Computational Data Types chapter in this manual for more details.



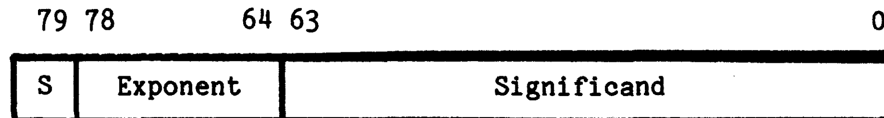
si SHORT INTEGER (16 bits)
These 16-bit operands represent signed integer values in the range -32,768 to 32,767 in 2's complement form.



so SHORT ORDINAL (16 bits)
These 16-bit operands represent unsigned integer values in the range 0 to 65,535, or bit strings of 16 bits or less.

sr SHORT REAL (32 bits)
These 32-bit operands represent floating-point numbers. See the Computational Data Types chapter in this manual for more details.

tr TEMPORARY REAL (80 bits)
These 80-bit operands represent floating-point numbers. See the Computational Data Types chapter in this manual for more details.



DATA OPERATORS

CHARACTER OPERATORS

During the execution of instructions using these character operators, if an arithmetic operation produces a result that cannot be represented in 8 bits, the operation is terminated without storing a result, and the Character Overflow Fault is raised. This occurs for any result < 0 or > 255. See the Fault and Trace Reference chapter for more details on faulting.

MOVE CHARACTER MOV_C

ID#	Operands 1	2	3
1	c	c	-

Opcode	Reference	Format	Class
00	varies	varies	011110

Character operand 1 is copied to character operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO CHARACTER ZRO_C

ID#	Operands 1	2	3
2	c	-	-

Opcode	Reference	Format	Class
0	varies	varies	010110

A character value of zero is stored in operand 1.

ONE CHARACTER ONE_C

ID#	Operands 1	2	3
3	c	-	-

Opcode	Reference	Format	Class
01	varies	varies	010110

A character value of one is stored in operand 1.

SAVE CHARACTER SAV_C

ID#	Operands 1	2	3
4	c	-	-

Opcode	Reference	Format	Class
11	varies	varies	010110

The character on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for operand 1 results in an operand stack in which the top two double-byte locations contain the same character value in the lower byte of each.

AND CHARACTER AND_C

ID#	Operands		
	1	2	3
5	c	c	c

Opcode	Reference	Format	Class
000	varies	varies	011101

Operand 1 is logically ANDed with operand 2. A bit in the result is set if the corresponding bits of both source operands are set; otherwise the result bit is cleared. The result is stored in operand 3.

INCLUSIVE OR CHARACTER IOR_C

ID#	Operands		
	1	2	3
6	c	c	c

Opcode	Reference	Format	Class
100	varies	varies	011101

Operand 1 is logically ORed (INCLUSIVE ORed) with operand 2. A bit in the result is set if either or both corresponding bits in the source operands are set; otherwise the result bit is cleared. The result is stored in operand 3.

EXCLUSIVE OR CHARACTER XOR_C

ID#	Operands		
	1	2	3
7	c	c	c

Opcode	Reference	Format	Class
010	varies	varies	011101

Operand 1 is logically XORed (EXCLUSIVE ORed) with operand 2. A bit in the result is set if the corresponding bits in the source operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared. The result is stored in operand 3. Note that applying this operator to two boolean operands produces a boolean result that is the same as if the two boolean operands are compared for inequality.

EQUIVALENCE CHARACTER EQV_C

ID#	Operands		
	1	2	3
8	c	c	c

Opcode	Reference	Format	Class
110	varies	varies	011101

Bitwise logical equivalence is performed between operand 1 and operand 2. A bit in the result is set if the corresponding bits in the source operands contain the same value (both are set or both are cleared); otherwise the result bit is cleared. The result is stored in operand 3. Note that applying this operator to two boolean operands produces a boolean result that is the same as if the two boolean operands are compared for equality.

NOT CHARACTER NOT_C

ID#	Operands 1	2	3
9	c	c	-

Opcode	Reference	Format	Class
10	varies	varies	011110

The bitwise logical NOT (1's complement) of character operand 1 is stored in character operand 2.

ADD CHARACTER ADD_C

ID#	Operands 1	2	3
10	c	c	c

Opcode	Reference	Format	Class
001	varies	varies	011101

Unsigned 8-bit addition is used to add operand 1 and operand 2. The result is stored in operand 3.

SUBTRACT CHARACTER SUB_C

ID#	Operands 1	2	3
11	c	c	c

Opcode	Reference	Format	Class
101	varies	varies	011101

Unsigned 8-bit subtraction is used to subtract operand 1 from operand 2. The result is stored in operand 3.

INCREMENT CHARACTER INC_C

ID#	Operands 1	2	3
12	c	c	-

Opcode	Reference	Format	Class
001	varies	varies	011110

Operand 1 is read and the value is incremented by one using unsigned 8-bit addition. The result is stored in operand 2.

DECREMENT CHARACTER DEC_C

ID#	Operands 1	2	3
13	c	c	-

Opcode	Reference	Format	Class
101	varies	varies	011110

Operand 1 is read and the value is decremented by one using unsigned 8-bit subtraction. The result is stored in operand 2.

EQUAL CHARACTER EQL_C

ID#	Operands 1	2	3
14	c	c	b

Opcode	Reference	Format	Class
0011	varies	varies	011101

An 8-bit comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3. Note that this operator should not be used to compare two boolean operands. The EQUIVALENCE CHARACTER operator should be used instead.

NOT EQUAL CHARACTER NEQ_C

ID#	Operands 1	2	3
15	c	c	b

Opcode	Reference	Format	Class
1011	varies	varies	011101

An 8-bit comparison is made between operand 1 and operand 2. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3. Note that this operator should not be used to compare two boolean operands for inequality. The EXCLUSIVE OR CHARACTER operator should be used instead.

EQUAL ZERO CHARACTER EQZ_C

ID#	Operands 1	2	3
16	c	b	-

Opcode	Reference	Format	Class
011	varies	varies	011110

An 8-bit comparison is made between operand 1 and a character value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2. Note that this operator should not be used to compare a boolean operand to a boolean value of FALSE. The NOT CHARACTER operator should be used instead.

NOT EQUAL ZERO CHARACTER NEZ_C

ID#	Operands 1	2	3
17	c	b	-

Opcode	Reference	Format	Class
111	varies	varies	011110

An 8-bit comparison is made between operand 1 and a character value of zero. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2. Note that this operator should not be used to compare a boolean operand to a boolean value of TRUE. The MOVE CHARACTER operator should be used instead.

LESS THAN CHARACTER LSS_C

ID#	Operands 1	2	3
18	c	c	b

Opcode	Reference	Format	Class
0111	varies	varies	011101

An unsigned 8-bit comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3. Note that this operator should not be used to compare booleans. The logical character operators should be used instead.

LESS THAN OR EQUAL CHARACTER LEQ_C

ID#	Operands 1	2	3
19	c	c	b

Opcode	Reference	Format	Class
1111	varies	varies	011101

An unsigned 8-bit comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3. Note that this operator should not be used to compare booleans. The logical character operators should be used instead.

CONVERT CHARACTER TO SHORT ORDINAL CVT_C_SO

ID#	Operands 1	2	3
20	c	so	-

Opcode	Reference	Format	Class
none	varies	varies	111110

Character operand 1 is converted to short-ordinal operand 2. Operand 1 is moved to the low-order byte of operand 2. The high-order byte of operand 2 is zeroed.

CONVERT CHARACTER TO INTEGER CVT_C_I

ID#	Operands 1	2	3
21	c	i	-

Opcode	Reference	Format	Class
none	varies	varies	000001

Character operand 1 is converted to integer operand 2. Operand 1 is moved to the low-order byte of operand 2. The three high-order bytes of operand 2 are zeroed.

SHORT-ORDINAL OPERATORS

During the execution of instructions using these short-ordinal operators, if any arithmetic operation produces a result that cannot be represented in 16 bits, the operation is terminated without storing a result, and the Short Ordinal Overflow Fault is raised. This occurs for any result < 0 or > 65,535. If the divisor is zero in any divide or remainder operation, the operation is suppressed, and the Short Ordinal Divide by Zero Fault is raised. See the Fault and Trace Reference chapter for more details on faulting.

MOVE SHORT ORDINAL MOV_SO

ID#	Operands 1	2	3
22	so	so	-

Opcode	Reference	Format	Class
0000	varies	varies	0100

Short-ordinal operand 1 is copied to short-ordinal operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO SHORT ORDINAL ZRO_SO

ID#	Operands 1	2	3
23	so	-	-

Opcode	Reference	Format	Class
000	varies	varies	0000

A short-ordinal value of zero is stored in operand 1.

ONE SHORT ORDINAL ONE_SO

ID#	Operands 1	2	3
24	so	-	-

Opcode	Reference	Format	Class
0100	varies	varies	0000

A short-ordinal value of one is stored in operand 1.

SAVE SHORT ORDINAL SAV_SO

ID#	Operands 1	2	3
25	so	-	-

Opcode	Reference	Format	Class
1100	varies	varies	0000

The short-ordinal on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for the operand 1 destination results in an operand stack in which the top two short-ordinal entries contain the same value.

AND SHORT ORDINAL AND_SO

ID#	Operands 1	2	3
26	so	so	so

Opcode	Reference	Format	Class
0000	varies	varies	0010

Operand 1 is logically ANDed with operand 2. A bit in the result is set if the corresponding bits of both source operands are set; otherwise the result bit is cleared. The result is stored in operand 3.

INCLUSIVE OR SHORT ORDINAL IOR_SO

ID#	Operands 1	2	3
27	so	so	so

Opcode	Reference	Format	Class
1000	varies	varies	0010

Operand 1 is logically ORed (INCLUSIVE ORed) with operand 2. A bit in the result is set if either or both corresponding bits in the source operands are set; otherwise the result bit is cleared. The result is stored in operand 3.

EXCLUSIVE OR SHORT ORDINAL XOR_SO

ID#	Operands 1	2	3
28	so	so	so

Opcode	Reference	Format	Class
0100	varies	varies	0010

Operand 1 is logically XORed (EXCLUSIVE ORed) with operand 2. A bit in the result is set if the corresponding bits in the source operands contain opposite values (one is set and the other is cleared); otherwise the result bit is cleared. The result is stored in operand 3.

EQUIVALENCE SHORT ORDINAL EQV_SO

ID#	Operands 1	2	3
29	so	so	so

Opcode	Reference	Format	Class
1100	varies	varies	0010

Bitwise logical equivalence is performed between operand 1 and operand 2. A bit in the result is set if the corresponding bits in the source operands contain the same value (both are set or both are cleared); otherwise the result bit is cleared. The result is stored in operand 3.

NOT SHORT ORDINAL

NOT_SO

ID#	Operands 1	2	3
30	so	so	-

Opcode	Reference	Format	Class
1000	varies	varies	0100

The bitwise logical NOT (1's complement) of short-ordinal operand 1 is stored in short-ordinal operand 2.

EXTRACT SHORT ORDINAL

EXT_SO

ID#	Operands 1	2	3
31	bfs	so	so

Opcode	Reference	Format	Class
0010	varies	varies	0010

Operand 2 is a short-ordinal from which a bit field is to be extracted. Operand 1 is a bit-field specifier that specifies the field to be extracted. The extracted bit field is right justified with high-order zeros to form a short-ordinal result that is stored in operand 3.

INSERT SHORT ORDINAL

INS_SO

ID#	Operands 1	2	3
32	bfs	so	so

Opcode	Reference	Format	Class
1010	varies	varies	0010

Operand 2 is a short-ordinal that contains a right-justified bit field to be inserted into the destination. Any high-order bits in operand 2 outside the bit field are ignored. Operand 1 is a bit-field specifier that specifies the field in the destination that is replaced by the inserted field. Operand 3 is the short-ordinal destination.

SIGNIFICANT BIT SHORT ORDINAL

SIG_SO

ID#	Operands 1	2	3
33	so	so	-

Opcode	Reference	Format	Class
0100	varies	varies	0100

The bit number (from 0 to 15) of the most-significant set bit in short-ordinal operand 1 is determined as a short-ordinal result and is stored in operand 2. If operand 1 has the value zero, the result is 16.

ADD SHORT ORDINAL ADD_SO

ID#	Operands 1	2	3
34	so	so	so

Opcode	Reference	Format	Class
0110	varies	varies	0010

Unsigned 16-bit addition is used to add operand 1 and operand 2. The result is stored in operand 3.

SUBTRACT SHORT ORDINAL SUB_SO

ID#	Operands 1	2	3
35	so	so	so

Opcode	Reference	Format	Class
1110	varies	varies	0010

Unsigned 16-bit subtraction is used to subtract operand 1 from operand 2. The result is stored in operand 3.

INCREMENT SHORT ORDINAL INC_SO

ID#	Operands 1	2	3
36	so	so	-

Opcode	Reference	Format	Class
1100	varies	varies	0100

Operand 1 is read and the value is incremented by one using unsigned 16-bit addition. The result is stored in operand 2.

DECREMENT SHORT ORDINAL DEC_SO

ID#	Operands 1	2	3
37	so	so	-

Opcode	Reference	Format	Class
0010	varies	varies	0100

Operand 1 is read and the value is decremented by one using unsigned 16-bit subtraction. The result is stored in operand 2.

MULTIPLY SHORT ORDINAL MUL_SO

ID#	Operands 1	2	3
38	so	so	so

Opcode	Reference	Format	Class
0001	varies	varies	0010

Unsigned 16-bit multiplication is used to multiply operand 1 and operand 2. The short-ordinal result is stored in operand 3.

DIVIDE SHORT ORDINAL — DIV_SO

ID#	Operands 1	2	3
39	so	so	so

Opcode	Reference	Format	Class
1001	varies	varies	0010

Unsigned 16-bit division is used to divide operand 1 into operand 2. The 16-bit quotient is stored in operand 3. Note that when the dividend is not an exact short-ordinal multiple of the divisor, the quotient is truncated toward zero (e.g., 8 divided by 3 yields 2).

REMAINDER SHORT ORDINAL — REM_SO

ID#	Operands 1	2	3
40	so	so	so

Opcode	Reference	Format	Class
0101	varies	varies	0010

Unsigned 16-bit division is used to divide operand 1 into operand 2. The 16-bit remainder is stored in operand 3. This operator performs the MOD function for the source operands.

EQUAL SHORT ORDINAL — EQL_SO

ID#	Operands 1	2	3
41	so	so	b

Opcode	Reference	Format	Class
000	varies	varies	111101

A 16-bit comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

NOT EQUAL SHORT ORDINAL — NEQ_SO

ID#	Operands 1	2	3
42	so	so	b

Opcode	Reference	Format	Class
100	varies	varies	111101

A 16-bit comparison is made between operand 1 and operand 2. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

EQUAL ZERO SHORT ORDINAL — EQZ_SO

ID#	Operands 1	2	3
43	so	b	-

Opcode	Reference	Format	Class
00	varies	varies	100001

A 16-bit comparison is made between operand 1 and a short-ordinal value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

NOT EQUAL ZERO SHORT ORDINAL NEZ_SO

ID#	Operands 1	2	3
44	so	b	-

Opcode	Reference	Format	Class
10	varies	varies	100001

A 16-bit comparison is made between operand 1 and a short-ordinal value of zero. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

LESS THAN SHORT ORDINAL LSS_SO

ID#	Operands 1	2	3
45	so	so	b

Opcode	Reference	Format	Class
010	varies	varies	111101

An unsigned 16-bit comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

LESS THAN OR EQUAL SHORT ORDINAL LEQ_SO

ID#	Operands 1	2	3
46	so	so	b

Opcode	Reference	Format	Class
110	varies	varies	111101

An unsigned 16-bit comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

CONVERT SHORT ORDINAL TO INTEGER CVT_SO_I

ID#	Operands 1	2	3
47	so	i	-

Opcode	Reference	Format	Class
00	varies	varies	010001

Short-ordinal operand 1 is converted to integer operand 2. Operand 1 is moved to the low-order 16 bits of operand 2. The high-order 16 bits of operand 2 are zeroed.

SHORT-INTEGER OPERATORS

During the execution of instructions using these short-integer operators, if any arithmetic operation produces a result that cannot be represented in a 16-bit 2's complement value, the operation is terminated without storing a result, and the Short Integer Overflow Fault is raised. This occurs for any result < -32,768 or > 32,767. If the divisor is zero in any division or remainder operation, the operation is suppressed, and the Short Integer Divide by Zero Fault is raised. See the Fault and Trace Reference chapter for more details on faulting.

MOVE SHORT INTEGER MOV_SI

ID#	Operands 1	Operands 2	Operands 3
22	si	si	-

Opcode	Reference	Format	Class
0000	varies	varies	0100

Short-integer operand 1 is copied to short-integer operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO SHORT INTEGER ZRO_SI

ID#	Operands 1	Operands 2	Operands 3
23	si	-	-

Opcode	Reference	Format	Class
000	varies	varies	0000

A short-integer value of zero is stored in operand 1.

ONE SHORT INTEGER ONE_SI

ID#	Operands 1	Operands 2	Operands 3
24	si	-	-

Opcode	Reference	Format	Class
0100	varies	varies	0000

A short-integer value of one is stored in operand 1.

SAVE SHORT INTEGER SAV_SI

ID#	Operands 1	Operands 2	Operands 3
25	si	-	-

Opcode	Reference	Format	Class
1100	varies	varies	0000

The short-integer on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for the operand 1 destination results in an operand stack in which the top two short-integer entries contain the same value.

ADD SHORT INTEGER — ADD_SI

ID#	Operands 1	2	3
48	si	si	si

Opcode	Reference	Format	Class
1101	varies	varies	0010

Signed 16-bit addition is used to add operand 1 and operand 2. The result is stored in operand 3.

SUBTRACT SHORT INTEGER — SUB_SI

ID#	Operands 1	2	3
49	si	si	si

Opcode	Reference	Format	Class
0011	varies	varies	0010

Signed 16-bit subtraction is used to subtract operand 1 from operand 2. The result is stored in operand 3.

INCREMENT SHORT INTEGER — INC_SI

ID#	Operands 1	2	3
50	si	si	-

Opcode	Reference	Format	Class
1010	varies	varies	0100

Operand 1 is read and the value is incremented by one using signed 16-bit addition. The result is stored in operand 2.

DECREMENT SHORT INTEGER — DEC_SI

ID#	Operands 1	2	3
51	si	si	-

Opcode	Reference	Format	Class
0110	varies	varies	0100

Operand 1 is read and the value is decremented by one using signed 16-bit subtraction. The result is stored in operand 2.

NEGATE SHORT INTEGER — NEG_SI

ID#	Operands 1	2	3
52	si	si	-

Opcode	Reference	Format	Class
1110	varies	varies	0100

The 2's complement of short-integer operand 1 is stored in short-integer operand 2.

MULTIPLY SHORT INTEGER MUL_SI

ID#	Operands 1	2	3
53	si	si	si

Opcode	Reference	Format	Class
01011	varies	varies	0010

Signed 16-bit multiplication is used to multiply operand 1 and operand 2. The short-integer result is stored in operand 3.

DIVIDE SHORT INTEGER DIV_SI

ID#	Operands 1	2	3
54	si	si	si

Opcode	Reference	Format	Class
11011	varies	varies	0010

Signed 16-bit division is used to divide operand 1 into operand 2. The 16-bit quotient is stored in operand 3. Note that when the dividend is not an exact short-integer multiple of the divisor, the quotient is truncated toward zero (e.g., 8 divided by 3 yields 2 and -8 divided by 3 yields -2).

REMAINDER SHORT INTEGER REM_SI

ID#	Operands 1	2	3
55	si	si	si

Opcode	Reference	Format	Class
00111	varies	varies	0010

Signed 16-bit division is used to divide operand 1 into operand 2. The signed 16-bit remainder is stored in operand 3. The sign of the remainder is the same as the sign of the dividend (operand 2). This operator performs the REM function for the source operands.

EQUAL SHORT INTEGER EQL_SI

ID#	Operands 1	2	3
41	si	si	b

Opcode	Reference	Format	Class
000	varies	varies	111101

A 16-bit comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

NOT EQUAL SHORT INTEGER NEQ_SI

ID#	Operands 1	2	3
42	si	si	b

Opcode	Reference	Format	Class
100	varies	varies	111101

A 16-bit comparison is made between operand 1 and operand 2. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

EQUAL ZERO SHORT INTEGER EQZ_SI

ID#	Operands 1	2	3
43	si	b	-

Opcode	Reference	Format	Class
00	varies	varies	100001

A 16-bit comparison is made between operand 1 and a short-integer value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

NOT EQUAL ZERO SHORT INTEGER NEZ_SI

ID#	Operands 1	2	3
44	si	b	-

Opcode	Reference	Format	Class
10	varies	varies	100001

A 16-bit comparison is made between operand 1 and a short-integer value of zero. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

LESS THAN SHORT INTEGER LSS_SI

ID#	Operands 1	2	3
56	si	si	b

Opcode	Reference	Format	Class
001	varies	varies	111101

A signed 16-bit comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

LESS THAN OR EQUAL SHORT INTEGER LEQ_SI

ID#	Operands 1	2	3
57	si	si	b

Opcode	Reference	Format	Class
101	varies	varies	111101

A signed 16-bit comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

POSITIVE SHORT INTEGER PTV_SI

ID#	Operands 1	2	3
58	si	b	-

Opcode	Reference	Format	Class
01	varies	varies	100001

If short-integer operand 1 is positive (greater than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2. Note that the result is FALSE if operand 1 is zero.

NEGATIVE SHORT INTEGER NTV_SI

ID#	Operands 1	2	3
59	si	b	-

Opcode	Reference	Format	Class
011	varies	varies	100001

If short-integer operand 1 is negative (less than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

MOVE IN RANGE SHORT INTEGER MIR_SI

ID#	Operands 1	2	3
60	pw	si	si

Opcode	Reference	Format	Class
none	varies	varies	110011

Operation: If short-integer operand 2 is within the range specified by operand 1, store the value of operand 2 in operand 3.

Operand 1: A packed word operand comprised of the following:
- 0 - 15: Contains the short-integer lower bound of the range
- 16 - 31: Contains the short-integer upper bound of the range

Operand 2: Contains the short-integer source operand

Operand 3: Contains the short-integer destination operand

Action: o If short-integer operand 2 is less than the lower bound, raise the Short-Integer Underflow fault. If operand 2 is greater than the upper bound, raise the Short-Integer Overflow fault.

Otherwise, store the value of operand 2 in operand 3.

CONVERT SHORT INTEGER TO INTEGER CVT_SI_I

ID#	Operands 1	2	3
61	si	i	–

Opcode	Reference	Format	Class
010	varies	varies	010001

Short-integer operand 1 is converted to integer operand 2. Operand 1 is moved to the low-order 16-bits of operand 2. The sign bit (bit 15) of operand 1 is extended to the high-order 16-bits of operand 2. (E.g., if the sign bit is 1, then the high-order 16 bits of operand 2 are all set to 1.)

ORDINAL OPERATORS

During the execution of instructions using these ordinal operators, if any arithmetic operation produces a result that cannot be represented in 32 bits, the operation is terminated without storing a result, and the Ordinal Overflow Fault is raised. This occurs for any result < 0 or > 4,294,967,295. If the divisor is zero in any divide or remainder operation, the operation is suppressed, and the Ordinal Divide by Zero Fault is raised. See the Fault and Trace Reference chapter for more details on faulting.

MOVE ORDINAL — MOV_O

ID#	Operands 1	2	3
62	o	o	-

Opcode	Reference	Format	Class
000	varies	varies	1100

Ordinal operand 1 is copied to ordinal operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO ORDINAL — ZRO_O

ID#	Operands 1	2	3
63	o	-	-

Opcode	Reference	Format	Class
00	varies	varies	110110

An ordinal value of zero is stored in operand 1.

ONE ORDINAL — ONE_O

ID#	Operands 1	2	3
64	o	-	-

Opcode	Reference	Format	Class
010	varies	varies	110110

An ordinal value of one is stored in operand 1.

SAVE ORDINAL — SAV_O

ID#	Operands 1	2	3
65	o	-	-

Opcode	Reference	Format	Class
110	varies	varies	110110

The ordinal on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for the operand 1 destination results in an operand stack in which the top two ordinal entries contain the same value.

AND ORDINAL AND_O

ID#	Operands 1	2	3
66	o	o	o

Opcode	Reference	Format	Class
000	varies	varies	1010

Operand 1 is logically ANDed with operand 2. A bit in the result is set if the corresponding bits of both source operands are set; otherwise the result bit is cleared. The result is stored in operand 3.

INCLUSIVE OR ORDINAL IOR_O

ID#	Operands 1	2	3
67	o	o	o

Opcode	Reference	Format	Class
0100	varies	varies	1010

Operand 1 is logically ORed (INCLUSIVE ORed) with operand 2. A bit in the result is set if either or both corresponding bits in the source operands are set; otherwise the result bit is cleared. The result is stored is operand 3.

EXCLUSIVE OR ORDINAL XOR_O

ID#	Operands 1	2	3
68	o	o	o

Opcode	Reference	Format	Class
1100	varies	varies	1010

Operand 1 is logically XORed (EXCLUSIVE ORed) with operand 2. A bit in the result is set if the corresponding bits in the source operands contain opposite values (one is set and the other is cleared); otherwise the result bit is cleared. The result is stored in operand 3.

EQUIVALENCE ORDINAL EQV_O

ID#	Operands 1	2	3
69	o	o	o

Opcode	Reference	Format	Class
0010	varies	varies	1010

Bitwise logical equivalence is performed between operand 1 and operand 2. A bit in the result is set if the corresponding bits in the source operands contain the same value (both are set or both are cleared); otherwise the result bit is cleared. The result is stored in operand 3.

NOT ORDINAL

NOT_O

ID#	Operands 1	2	3
70	o	o	-

Opcode	Reference	Format	Class
100	varies	varies	1100

The bitwise logical NOT (1's complement) of ordinal operand 1 is stored in ordinal operand 2.

EXTRACT ORDINAL

EXT_O

ID#	Operands 1	2	3
71	bfs	o	o

Opcode	Reference	Format	Class
00	varies	varies	100011

Operand 2 is an ordinal from which a bit field is to be extracted. Operand 1 is a bit-field specifier that specifies the field to be extracted. The extracted bit field is right justified with high-order zeros to form an ordinal result that is stored in operand 3.

INSERT ORDINAL

INS_O

ID#	Operands 1	2	3
72	bfs	o	o

Opcode	Reference	Format	Class
10	varies	varies	100011

Operand 2 is an ordinal that contains a right-justified bit field to be inserted into the destination. Any high-order bits in operand 2 outside the bit field are ignored. Operand 1 is a bit-field specifier that specifies the field in the destination that is replaced by the inserted field. Operand 3 is the ordinal destination.

SIGNIFICANT BIT ORDINAL

SIG_O

ID#	Operands 1	2	3
73	o	so	-

Opcode	Reference	Format	Class
00	varies	varies	101001

The bit number (from 0 to 31) of the most-significant set bit in ordinal operand 1 is determined as a short-ordinal result and is stored in operand 2. If operand 1 has the value zero, the result is 32.

ADD ORDINAL

ADD_O

ID#	Operands 1	2	3
74	o	o	o

Opcode	Reference	Format	Class
1010	varies	varies	1010

Unsigned 32-bit addition is used to add operand 1 and operand 2. The result is stored in operand 3.

SUBTRACT ORDINAL SUB_O

ID#	Operands 1	2	3
75	o	o	o

Opcode	Reference	Format	Class
0110	varies	varies	1010

Unsigned 32-bit subtraction is used to subtract operand 1 from operand 2. The result is stored in operand 3.

INCREMENT ORDINAL INC_O

ID#	Operands 1	2	3
76	o	o	-

Opcode	Reference	Format	Class
010	varies	varies	1100

Operand 1 is read and the value is incremented by one using unsigned 32-bit addition. The result is stored in operand 2.

DECREMENT ORDINAL DEC_O

ID#	Operands 1	2	3
77	o	o	-

Opcode	Reference	Format	Class
0110	varies	varies	1100

Operand 1 is read and the value is decremented by one using unsigned 32-bit subtraction. The result is stored in operand 2.

MULTIPLY ORDINAL MUL_O

ID#	Operands 1	2	3
78	o	o	o

Opcode	Reference	Format	Class
1110	varies	varies	1010

Unsigned 32-bit multiplication is used to multiply operand 1 and operand 2. The ordinal result is stored in operand 3.

DIVIDE ORDINAL DIV_O

ID#	Operands 1	2	3
79	o	o	o

Opcode	Reference	Format	Class
0001	varies	varies	1010

Unsigned 32-bit division is used to divide operand 1 into operand 2. The 32-bit quotient is stored in operand 3. Note that when the dividend is not an exact ordinal multiple of the divisor, the quotient is truncated toward zero (e.g., 8 divided by 3 yields 2).

REMAINDER ORDINAL REM_O

ID#	Operands 1	2	3
80	o	o	o

Opcode	Reference	Format	Class
1001	varies	varies	1010

Unsigned 32-bit division is used to divide operand 1 into operand 2. The 32-bit remainder is stored in operand 3. This operator performs the REM function for the source operands.

INDEX ORDINAL IDX_O

ID#	Operands 1	2	3
81	pw	o	pw

Opcode	Reference	Format	Class
0101	varies	varies	1010

Operation: Computes the access selector and displacement for an element of a large multi-segment (2K bytes/segment) array.

Operand 1: A packed word operand comprised of the following:

0 - 15: Contains a scale factor that specifies the size of an array element in bytes. Only the least-significant 4 bits are interpreted. The size of each array element (in bytes) is two raised to the power of this 4-bit value. For example, a value of 2 is for a 4-byte size, a value of 3 is for an 8-byte size, etc. The size of an array element is then used to scale the index (operand 2).

16 - 31: Contains the access selector for the first segment of the multi-segment array.

Operand 2: Contains an ordinal index into the multi-segment array

Operand 3: A packed word destination comprised of the following:

0 - 15: The computed byte displacement into the selected segment

16 - 31: The computed access selector for the array segment in which the indexed element is located

Action:

- Scale the index (operand 2) by the scale factor specified by operand 1.
- Extract the least-significant 11 bits of the scaled index. This is the computed displacement (zero extended) into the selected 2K segment.
- Bits 12 through 25 of the scaled index are extracted and added to the access index of the base access selector specified in operand 1 to form a computed access selector.
- Store the computed displacement and computed access selector into their respective locations in operand 3.

Notes:

- Bits 26 through 31 of the scaled index are ignored.
- The access part containing ADs for the array segments must be entered as an environment. The ADs for the 2Kbyte segments should be contiguous, so that successive access index values select successive segments.

EQUAL ORDINAL EQL_O

ID#	Operands 1	2	3
82	o	o	b

Opcode	Reference	Format	Class
000	varies	varies	001011

A 32-bit comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

NOT EQUAL ORDINAL NEQ_O

ID#	Operands 1	2	3
83	o	o	b

Opcode	Reference	Format	Class
100	varies	varies	001011

A 32-bit comparison is made between operand 1 and operand 2. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

EQUAL ZERO ORDINAL EQZ_O

ID#	Operands 1	2	3
84	o	b	-

Opcode	Reference	Format	Class
000	varies	varies	001001

A 32-bit comparison is made between operand 1 and an ordinal value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

NOT EQUAL ZERO ORDINAL NEZ_O

ID#	Operands 1	2	3
85	o	b	-

Opcode	Reference	Format	Class
100	varies	varies	001001

A 32-bit comparison is made between operand 1 and an ordinal value of zero. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

LESS THAN ORDINAL LSS_O

ID#	Operands 1	2	3
86	o	o	b

Opcode	Reference	Format	Class
010	varies	varies	001011

An unsigned 32-bit comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

LESS THAN OR EQUAL TO ORDINAL LEQ_O

ID#	Operands 1	2	3
87	o	o	b

Opcode	Reference	Format	Class
110	varies	varies	001011

An unsigned 32-bit comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

CONVERT ORDINAL TO INTEGER CVT_O_I

ID#	Operands 1	2	3
88	o	i	-

Opcode	Reference	Format	Class
1110	varies	varies	1100

Ordinal operand 1 is converted to integer operand 2. If the most-significant bit of operand 1 has the value 1 (operand 1 > 32,767), the Integer Overflow Fault is raised.

CONVERT ORDINAL TO TEMPORARY REAL CVT_O_TR

ID#	Operands 1	2	3
89	o	tr	-

Opcode	Reference	Format	Class
0	varies	varies	011001

Ordinal operand 1 is converted exactly to temporary-real operand 2. The settings of the Rounding Control bits and of the Precision Control bits have no effect on the value of the result.

INTEGER OPERATORS

During the execution of instructions using these integer operators, if any arithmetic instruction produces a result that cannot be represented in a 32-bit 2's complement value, the operation is terminated without storing a result, and the Integer Overflow Fault is raised. This occurs for any result < -2,147,483,648 or > 2,147,483,647. If the divisor is zero in any divide or remainder operation, the operation is suppressed, and the Integer Divide by Zero Fault is raised. See the Fault and Trace Reference chapter for more details on faulting.

MOVE INTEGER   MOV_I

ID#	Operands 1	2	3
62	i	i	-

Opcode	Reference	Format	Class
000	varies	varies	1100

Integer operand 1 is copied to integer operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO INTEGER   ZRO_I

ID#	Operands 1	2	3
63	i	-	-

Opcode	Reference	Format	Class
00	varies	varies	110110

An integer value of zero is stored in operand 1.

ONE INTEGER   ONE_I

ID#	Operands 1	2	3
64	i	-	-

Opcode	Reference	Format	Class
010	varies	varies	110110

An integer value of one is stored in operand 1.

SAVE INTEGER   SAV_I

ID#	Operands 1	2	3
65	i	-	-

Opcode	Reference	Format	Class
110	varies	varies	110110

The integer on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for the operand 1 destination results in an operand stack in which the top two integer entries contain the same value.

ADD INTEGER ADD_I

ID#	Operands		
	1	2	3
90	i	i	i

Opcode	Reference	Format	Class
1101	varies	varies	1010

Signed 32-bit addition is used to add operand 1 and operand 2. The result is stored in operand 3.

SUBTRACT INTEGER SUB_I

ID#	Operands		
	1	2	3
91	i	i	i

Opcode	Reference	Format	Class
0011	varies	varies	1010

Signed 32-bit subtraction is used to subtract operand 1 from operand 2. The result is stored in operand 3.

INCREMENT INTEGER INC_I

ID#	Operands		
	1	2	3
92	i	i	-

Opcode	Reference	Format	Class
0001	varies	varies	1100

Operand 1 is read and the value is incremented by one using signed 32-bit addition. The result is stored in operand 2.

DECREMENT INTEGER DEC_I

ID#	Operands		
	1	2	3
93	i	i	-

Opcode	Reference	Format	Class
1001	varies	varies	1100

Operand 1 is read and the value is decremented by one using signed 32-bit subtraction. The result is stored in operand 2.

NEGATE INTEGER NEG_I

ID#	Operands		
	1	2	3
94	i	i	-

Opcode	Reference	Format	Class
0101	varies	varies	1100

The 2's complement of integer operand 1 is stored in integer operand 2.

MULTIPLY INTEGER MUL_I

ID#	Operands 1	2	3
95	i	i	i

Opcode	Reference	Format	Class
1011	varies	varies	1010

Signed 32-bit multiplication is used to multiply operand 1 and operand 2. The integer result is stored in operand 3.

DIVIDE INTEGER DIV_I

ID#	Operands 1	2	3
96	i	i	i

Opcode	Reference	Format	Class
0111	varies	varies	1010

Signed 32-bit division is used to divide operand 1 into operand 2. The 32-bit quotient is stored in operand 3. Note that when the dividend is not an exact integer multiple of the divisor, the quotient is truncated toward zero (e.g., 8 divided by 3 yields 2 and -8 divided by 3 yields -2).

REMAINDER INTEGER REM_I

ID#	Operands 1	2	3
97	i	i	i

Opcode	Reference	Format	Class
1111	varies	varies	1010

Signed 32-bit division is used to divide operand 1 into operand 2. The signed 32-bit remainder is stored in operand 3. The sign of the remainder is the same as the sign of the dividend (operand 2). This operator performs the REM function for the source operands.

EQUAL INTEGER EQL_I

ID#	Operands 1	2	3
82	i	i	b

Opcode	Reference	Format	Class
000	varies	varies	001011

A 32-bit comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

NOT EQUAL INTEGER

NEQ_I

ID#	Operands 1	2	3
83	i	i	b

Opcode	Reference	Format	Class
100	varies	varies	001011

A 32-bit comparison is made between operand 1 and operand 2. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

EQUAL ZERO INTEGER

EQZ_I

ID#	Operands 1	2	3
84	i	b	-

Opcode	Reference	Format	Class
000	varies	varies	001001

A 32-bit comparison is made between operand 1 and an integer value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

NOT EQUAL ZERO INTEGER

NEZ_I

ID#	Operands 1	2	3
85	i	b	-

Opcode	Reference	Format	Class
100	varies	varies	001001

A 32-bit comparison is made between operand 1 and an integer value of zero. If they are not equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

LESS THAN INTEGER

LSS_I

ID#	Operands 1	2	3
98	i	i	b

Opcode	Reference	Format	Class
001	varies	varies	001011

A signed 32-bit comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

LESS THAN OR EQUAL INTEGER LEQ_I

ID#	Operands 1	2	3
99	i	i	b

Opcode	Reference	Format	Class
101	varies	varies	001011

A signed 32-bit comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

POSITIVE INTEGER PTV_I

ID#	Operands 1	2	3
100	i	b	-

Opcode	Reference	Format	Class
010	varies	varies	001001

If integer operand 1 is positive (greater than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2. Note that the result is FALSE if operand 1 is zero.

NEGATIVE INTEGER NTV_I

ID#	Operands 1	2	3
101	i	b	-

Opcode	Reference	Format	Class
110	varies	varies	001001

If integer operand 1 is negative (less than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

MOVE IN RANGE INTEGER MIR_I

ID#	Operands 1	2	3
102	pdw	i	i

Opcode	Reference	Format	Class
none	varies	varies	111011

Operation: If integer operand 2 is within the range specified by operand 1, store the value of operand 2 in operand 3.

Operand 1: A packed double word comprised of the following:
0 - 31: Contains the integer lower bound of the range
32 - 63: Contains the integer upper bound of the range

Operand 2: Contains the integer source operand

Operand 3: Contains the integer destination operand

Action:

- If integer operand 2 is less than the lower bound, raise the Integer Underflow Fault. If integer operand 2 is greater than the upper bound, raise the Integer Overflow Fault.
- Otherwise, store the value of operand 2 in operand 3.

CONVERT INTEGER TO CHARACTER CVT_I_C

ID#	Operands		
	1	2	3
103	i	c	-

Opcode	Reference	Format	Class
001	varies	varies	001001

Integer operand 1 is converted to character operand 2. If operand 1 is < 0 or > 255, the Character Overflow Fault is raised.

CONVERT INTEGER TO SHORT ORDINAL CVT_I_SO

ID#	Operands		
	1	2	3
104	i	so	-

Opcode	Reference	Format	Class
10	varies	varies	101001

Integer operand 1 is converted to short-ordinal operand 2. If operand 1 is < 0 or > 65,535, the Short-Ordinal Overflow Fault is raised.

CONVERT INTEGER TO SHORT INTEGER CVT_I_SI

ID#	Operands		
	1	2	3
105	i	si	-

Opcode	Reference	Format	Class
01	varies	varies	101001

Integer operand 1 is converted to short-integer operand 2. If operand 1 is < -32,768 or > 32,767, the Short-Integer Overflow Fault is raised.

CONVERT INTEGER TO ORDINAL CVT_I_O

ID#	Operands		
	1	2	3
88	i	o	-

Opcode	Reference	Format	Class
1110	varies	varies	1100

Integer operand 1 is converted to ordinal operand 2. If operand 1 is negative, the Ordinal Overflow fault is raised.

CONVERT INTEGER TO TEMPORARY REAL CVT_I_TR

ID#	Operands		
	1	2	3
106	i	tr	-

Opcode	Reference	Format	Class
01	varies	varies	011001

Integer operand 1 is converted exactly to temporary-real operand 2. The settings of the Rounding Control bits and of the Precision Control bits have no effect on the value of the result.

SHORT-REAL OPERATORS

During the execution of instructions using these short-real operators, if rounding is required to produce the final result, the type of rounding used is determined by the setting of the Rounding Control bits (in the Context Status field). Where noted in the operator descriptions, the precision maintained in temporary-real results is determined by the setting of the Precision Control bits (in the Context Status field).

The following data operator faults are recognized by the processor during short-real instructions: Overflow, Underflow, Inexact, and Domain Error. See the Fault and Trace Reference chapter for details about which of these faults can be raised by specific short-real instructions.

MOVE SHORT REAL MOV_SR

ID#	Operands 1	2	3
62	sr	sr	-

Opcode	Reference	Format	Class
000	varies	varies	1100

Short-real operand 1 is copied to short-real operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO SHORT REAL ZRO_SR

ID#	Operands 1	2	3
63	sr	-	-

Opcode	Reference	Format	Class
00	varies	varies	110110

A short-real value of zero is stored in operand 1.

SAVE SHORT REAL SAV_SR

ID#	Operands 1	2	3
65	sr	-	-

Opcode	Reference	Format	Class
110	varies	varies	110110

The short-real on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for the operand 1 destination results in an operand stack in which the top two short-real entries contain the same value.

ADD SHORT REAL

ADD_SR

ID#	Operands 1	2	3
107	sr	sr	tr

Opcode	Reference	Format	Class
00	varies	varies	101011

Short-real operand 1 is added to short-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

ADD TEMPORARY REAL TO SHORT REAL

ADD_TR_SR

ID#	Operands 1	2	3
108	tr	sr	tr

Opcode	Reference	Format	Class
00	varies	varies	110111

Temporary-real operand 1 is added to short-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

ADD SHORT REAL TO TEMPORARY REAL

ADD_SR_TR

ID#	Operands 1	2	3
109	sr	tr	tr

Opcode	Reference	Format	Class
00	varies	varies	011011

Short-real operand 1 is added to temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

SUBTRACT SHORT REAL

SUB_SR

ID#	Operands 1	2	3
110	sr	sr	tr

Opcode	Reference	Format	Class
10	varies	varies	101011

Short-real operand 1 is subtracted from short-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

SUBTRACT TEMPORARY REAL FROM SHORT REAL SUB_TR_SR

ID#	Operands 1	2	3
111	tr	sr	tr

Opcode	Reference	Format	Class
10	varies	varies	110111

Temporary-real operand 1 is subtracted from short-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

SUBTRACT SHORT REAL FROM TEMPORARY REAL SUB_SR_TR

ID#	Operands 1	2	3
112	sr	tr	tr

Opcode	Reference	Format	Class
10	varies	varies	011011

Short-real operand 1 is subtracted from temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

MULTIPLY SHORT REAL MUL_SR

ID#	Operands 1	2	3
113	sr	sr	tr

Opcode	Reference	Format	Class
01	varies	varies	101011

Short-real operand 1 is multiplied by short-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

MULTIPLY TEMPORARY REAL BY SHORT REAL MUL_TR_SR

ID#	Operands 1	2	3
114	tr	sr	tr

Opcode	Reference	Format	Class
01	varies	varies	110111

Temporary-real operand 1 is multiplied by short-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

MULTIPLY SHORT REAL BY TEMPORARY REAL MUL_SR_TR

ID#	Operands 1	2	3
115	sr	tr	tr

Opcode	Reference	Format	Class
01	varies	varies	011011

Short-real operand 1 is multiplied by temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

DIVIDE SHORT REAL DIV_SR

ID#	Operands 1	2	3
116	sr	sr	tr

Opcode	Reference	Format	Class
11	varies	varies	101011

Short-real operand 1 is divided into short-real operand 2 to produce a quotient that is stored in temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

DIVIDE TEMPORARY REAL INTO SHORT REAL DIV_TR_SR

ID#	Operands 1	2	3
117	tr	sr	tr

Opcode	Reference	Format	Class
11	varies	varies	110111

Temporary-real operand 1 is divided into short-real operand 2 to produce a quotient that is stored in temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

DIVIDE SHORT REAL INTO TEMPORARY REAL DIV_SR_TR

ID#	Operands 1	2	3
118	sr	tr	tr

Opcode	Reference	Format	Class
11	varies	varies	011011

Short-real operand 1 is divided into temporary-real operand 2 to produce a quotient that is stored in temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

NEGATE SHORT REAL NEG_SR

ID#	Operands 1	2	3
119	sr	sr	-

Opcode	Reference	Format	Class
1101	varies	varies	1100

The negated value of short-real operand 1 is stored in short-real operand 2.

ABSOLUTE VALUE SHORT REAL ABS_SR

ID#	Operands 1	2	3
120	sr	sr	-

Opcode	Reference	Format	Class
0011	varies	varies	1100

The absolute value of short-real operand 1 is stored in short-real operand 2.

EQUAL SHORT REAL EQL_SR

ID#	Operands 1	2	3
121	sr	sr	b

Opcode	Reference	Format	Class
011	varies	varies	001011

A short-real comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

EQUAL ZERO SHORT REAL EQZ_SR

ID#	Operands 1	2	3
122	sr	b	-

Opcode	Reference	Format	Class
101	varies	varies	001001

A short-real comparison is made between operand 1 and a short-real value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

LESS THAN SHORT REAL LSS_SR

ID#	Operands 1	2	3
123	sr	sr	b

Opcode	Reference	Format	Class
0111	varies	varies	001011

A short-real comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

LESS THAN OR EQUAL SHORT REAL LEQ_SR

ID#	Operands 1	2	3
124	sr	sr	b

Opcode	Reference	Format	Class
1111	varies	varies	001011

A short-real comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

POSITIVE SHORT REAL PTV_SR

ID#	Operands 1	2	3
125	sr	b	-

Opcode	Reference	Format	Class
011	varies	varies	001001

If short-real operand 1 is positive (greater than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2. Note that the result is FALSE if operand 1 is zero.

NEGATIVE SHORT REAL NTV_SR

ID#	Operands 1	2	3
126	sr	b	-

Opcode	Reference	Format	Class
111	varies	varies	001001

If short-real operand 1 is negative (less than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

CONVERT SHORT REAL TO TEMPORARY REAL CVT_SR_TR

ID#	Operands 1	2	3
127	sr	tr	-

Opcode	Reference	Format	Class
11	varies	varies	011001

Short-real operand 1 is converted without loss of precision to temporary-real operand 2. The settings of the Rounding Control bits and of the Precision Control bits have no effect on the value of the result.

REAL OPERATORS

During the execution of instructions using these real operators, if rounding is required to produce the final result, the type of rounding used is determined by the setting of the Rounding Control bits (in the Context Status field). Where noted in the operator descriptions, the precision maintained in temporary-real results is determined by the setting of the Precision Control bits (in the Context Status field).

The following data operator faults are recognized by the processor during these real instructions: Overflow, Underflow, Inexact, and Domain Error. See the Fault and Trace Reference chapter for details about which of these faults can be raised by specific real instructions.

MOVE REAL MOV_R

ID#	Operands 1	2	3
128	r	r	-

Opcode	Reference	Format	Class
00	varies	varies	000101

Real operand 1 is copied to real operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO REAL ZRO_R

ID#	Operands 1	2	3
129	r	-	-

Opcode	Reference	Format	Class
0	varies	varies	001110

A real value of zero is stored in real operand 1.

SAVE REAL SAV_R

ID#	Operands 1	2	3
130	r	-	-

Opcode	Reference	Format	Class
1	varies	varies	001110

The real operand on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for the operand 1 destination results in an operand stack in which the top two real entries contain the same value.

ADD REAL ADD_R

ID#	Operands 1	Operands 2	Operands 3
131	r	r	tr

Opcode	Reference	Format	Class
00	varies	varies	100111

Real operand 1 is added to real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

ADD TEMPORARY REAL TO REAL ADD_TR_R

ID#	Operands 1	Operands 2	Operands 3
132	tr	r	tr

Opcode	Reference	Format	Class
00	varies	varies	001111

Temporary-real operand 1 is added to real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

ADD REAL TO TEMPORARY REAL ADD_R_TR

ID#	Operands 1	Operands 2	Operands 3
133	r	tr	tr

Opcode	Reference	Format	Class
00	varies	varies	010111

Real operand 1 is added to temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

SUBTRACT REAL SUB_R

ID#	Operands 1	Operands 2	Operands 3
134	r	r	tr

Opcode	Reference	Format	Class
10	varies	varies	100111

Real operand 1 is subtracted from real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

SUBTRACT TEMPORARY REAL FROM REAL SUB_TR_R

ID#	Operands 1	Operands 2	Operands 3
135	tr	r	tr

Opcode	Reference	Format	Class
10	varies	varies	001111

Temporary-real operand 1 is subtracted from real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

SUBTRACT REAL FROM TEMPORARY REAL SUB_R_TR

ID#	Operands 1	Operands 2	Operands 3
136	r	tr	tr

Opcode	Reference	Format	Class
10	varies	varies	010111

Real operand 1 is subtracted from temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

MULTIPLY REAL MUL_R

ID#	Operands 1	Operands 2	Operands 3
137	r	r	tr

Opcode	Reference	Format	Class
01	varies	varies	100111

Real operand 1 is multiplied by real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

MULTIPLY TEMPORARY REAL BY REAL MUL_TR_R

ID#	Operands 1	Operands 2	Operands 3
138	tr	r	tr

Opcode	Reference	Format	Class
01	varies	varies	001111

Temporary-real operand 1 is multiplied by real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

MULTIPLY REAL BY TEMPORARY REAL MUL_R_TR

ID#	Operands 1	2	3
139	r	tr	tr

Opcode	Reference	Format	Class
01	varies	varies	010111

Real operand 1 is multiplied by temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

DIVIDE REAL DIV_R

ID#	Operands 1	2	3
140	r	r	tr

Opcode	Reference	Format	Class
11	varies	varies	100111

Real operand 1 is divided into real operand 2 to produce a quotient that is stored in temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

DIVIDE TEMPORARY REAL INTO REAL DIV_TR_R

ID#	Operands 1	2	3
141	tr	r	tr

Opcode	Reference	Format	Class
11	varies	varies	001111

Temporary-real operand 1 is divided into real operand 2 to produce a quotient that is stored in temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

DIVIDE REAL INTO TEMPORARY REAL DIV_R_TR

ID#	Operands 1	2	3
142	r	tr	tr

Opcode	Reference	Format	Class
11	varies	varies	010111

Real operand 1 is divided into temporary-real operand 2 to produce a quotient that is stored in temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

NEGATE REAL

NEG_R

ID#	Operands 1	Operands 2	Operands 3
143	r	r	-

Opcode	Reference	Format	Class
10	varies	varies	000101

The negated value of real operand 1 is stored in real operand 2.

ABSOLUTE VALUE REAL

ABS_R

ID#	Operands 1	Operands 2	Operands 3
144	r	r	-

Opcode	Reference	Format	Class
01	varies	varies	000101

The absolute value of real operand 1 is stored in real operand 2.

EQUAL REAL

EQL_R

ID#	Operands 1	Operands 2	Operands 3
145	r	r	b

Opcode	Reference	Format	Class
0	varies	varies	000111

A real comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

EQUAL ZERO REAL

EQZ_R

ID#	Operands 1	Operands 2	Operands 3
146	r	b	-

Opcode	Reference	Format	Class
0	varies	varies	111001

A real comparison is made between operand 1 and a real value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

LESS THAN REAL

LSS_R

ID#	Operands 1	Operands 2	Operands 3
147	r	r	b

Opcode	Reference	Format	Class
01	varies	varies	000111

A real comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

LESS THAN OR EQUAL REAL — LEQ_R

ID#	Operands 1	Operands 2	Operands 3
148	r	r	b

Opcode	Reference	Format	Class
11	varies	varies	000111

A real comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

POSITIVE REAL — PTV_R

ID#	Operands 1	Operands 2	Operands 3
149	r	b	-

Opcode	Reference	Format	Class
01	varies	varies	111001

If real operand 1 is positive (greater than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2. Note that the result is FALSE if operand 1 is zero.

NEGATIVE REAL — NTV_R

ID#	Operands 1	Operands 2	Operands 3
150	r	b	-

Opcode	Reference	Format	Class
11	varies	varies	111001

If real operand 1 is negative (less than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

CONVERT REAL TO TEMPORARY REAL — CVT_R_TR

ID#	Operands 1	Operands 2	Operands 3
151	r	tr	-

Opcode	Reference	Format	Class
none	varies	varies	100101

Real operand 1 is converted without loss of precision to temporary-real operand 2. The settings of the Rounding Control bits and of the Precision Control bits have no effect on the value of the result.

TEMPORARY-REAL OPERATORS

During the execution of instructions using these temporary-real operators, if rounding is required to produce the final result, the type of rounding used is determined by the setting of the Rounding Control bits (in the Context Status field). Where noted in the operator descriptions, the precision maintained in temporary-real results is determined by the setting of the Precision Control bits (in the Context Status field).

The following data operator faults are recognized by the processor during these temporary-real instructions: Overflow, Underflow, Inexact, and Domain Error. See the Fault and Trace Reference chapter for details about which of these faults can be raised by specific temporary-real instructions.

MOVE TEMPORARY REAL — MOV_TR

ID#	Operands 1	2	3
152	tr	tr	-

Opcode	Reference	Format	Class
00	varies	varies	101101

Temporary-real operand 1 is copied to temporary-real operand 2. Using the operand stack for operand 1 results in the classical POP stack operation, and using the operand stack for operand 2 results in the classical PUSH stack operation. Using the operand stack as both operand 1 and operand 2 results in no change.

ZERO TEMPORARY REAL — ZRO_TR

ID#	Operands 1	2	3
153	tr	-	-

Opcode	Reference	Format	Class
0	varies	varies	101110

A temporary-real value of zero is stored in operand 1.

SAVE TEMPORARY REAL — SAV_TR

ID#	Operands 1	2	3
154	tr	-	-

Opcode	Reference	Format	Class
1	varies	varies	101110

The temporary-real operand on top of the operand stack is read, without adjusting the stack, and copied to operand 1. Using the operand stack for the operand 1 destination results in an operand stack in which the top two temporary-real operands contain the same value.

ADD TEMPORARY REAL ADD_TR

ID#	Operands 1	Operands 2	Operands 3
155	tr	tr	tr

Opcode	Reference	Format	Class
00	varies	varies	011111

Temporary-real operand 1 is added to temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

SUBTRACT TEMPORARY REAL SUB_TR

ID#	Operands 1	Operands 2	Operands 3
156	tr	tr	tr

Opcode	Reference	Format	Class
10	varies	varies	011111

Temporary-real operand 1 is subtracted from temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

MULTIPLY TEMPORARY REAL MUL_TR

ID#	Operands 1	Operands 2	Operands 3
157	tr	tr	tr

Opcode	Reference	Format	Class
01	varies	varies	011111

Temporary-real operand 1 is multiplied by temporary-real operand 2 to produce temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

DIVIDE TEMPORARY REAL DIV_TR

ID#	Operands 1	Operands 2	Operands 3
158	tr	tr	tr

Opcode	Reference	Format	Class
011	varies	varies	011111

Temporary-real operand 1 is divided into temporary-real operand 2 to produce a quotient that is stored in temporary-real operand 3. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

REMAINDER TEMPORARY REAL

REM_TR

ID#	Operands 1	2	3
159	tr	tr	tr

Opcode	Reference	Format	Class
111	varies	varies	011111

Division is performed using temporary-real operand 1 as the divisor and temporary-real operand 2 as the dividend to produce a temporary-real partial remainder. Execution of this operator causes one step of the division algorithm to be performed. It can be iterated until a fixed number of division steps have been performed or until a partial remainder whose absolute value is less than the absolute value of the divisor is generated. In the latter case, that partial remainder is the true remainder of the division operation. If the result is not the true remainder, no rounding is done. In either case the partial remainder generated by the last division step is stored in temporary-real operand 3 with the same sign as that of operand 2. The remainder or partial remainder generated is always exact. This operator performs only the inner loop operation of the remainder function. See the Computational Data Types chapter in Part One of this manual for more information about how to calculate the true temporary-real remainder.

NEGATE TEMPORARY REAL

NEG_TR

ID#	Operands 1	2	3
160	tr	tr	-

Opcode	Reference	Format	Class
10	varies	varies	101101

The negated value of temporary-real operand 1 is stored in temporary-real operand 2.

SQUARE ROOT TEMPORARY REAL

SQT_TR

ID#	Operands 1	2	3
161	tr	tr	-

Opcode	Reference	Format	Class
01	varies	varies	101101

The square root of temporary-real operand 1 is computed and stored in temporary-real operand 2. The settings of the Rounding Control bits and the Precision Control bits specify the type of rounding that is used and the precision to which the result is rounded.

ABSOLUTE VALUE TEMPORARY REAL

ABS_TR

ID#	Operands 1	2	3
162	tr	tr	-

Opcode	Reference	Format	Class
11	varies	varies	101101

The absolute value of temporary-real operand 1 is stored in temporary-real operand 2.

EQUAL TEMPORARY REAL EQL_TR

ID#	Operands 1	2	3
163	tr	tr	b

Opcode	Reference	Format	Class
0	varies	varies	101111

A temporary-real comparison is made between operand 1 and operand 2. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

EQUAL ZERO TEMPORARY REAL EQZ_TR

ID#	Operands 1	2	3
164	tr	b	-

Opcode	Reference	Format	Class
0	varies	varies	010101

A temporary-real comparison is made between operand 1 and a temporary-real value of zero. If they are equal, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

LESS THAN TEMPORARY REAL LSS_TR

ID#	Operands 1	2	3
165	tr	tr	b

Opcode	Reference	Format	Class
01	varies	varies	101111

A temporary-real comparison is made between operand 1 and operand 2. If operand 1 is less than operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

LESS THAN OR EQUAL TEMPORARY REAL LEQ_TR

ID#	Operands 1	2	3
166	tr	tr	b

Opcode	Reference	Format	Class
11	varies	varies	101111

A temporary-real comparison is made between operand 1 and operand 2. If operand 1 is less than or equal to operand 2, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 3.

POSITIVE TEMPORARY REAL PTV_TR

ID#	Operands 1	2	3
167	tr	b	-

Opcode	Reference	Format	Class
01	varies	varies	010101

If temporary-real operand 1 is positive (greater than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2. Note that the result is FALSE if operand 1 is zero.

NEGATIVE TEMPORARY REAL NTV_TR

ID#	Operands 1	2	3
168	tr	b	-

Opcode	Reference	Format	Class
11	varies	varies	010101

If temporary-real operand 1 is negative (less than zero), the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in operand 2.

CONVERT TEMPORARY REAL TO ORDINAL CVT_TR_O

ID#	Operands 1	2	3
169	tr	o	-

Opcode	Reference	Format	Class
0	varies	varies	110101

Temporary-real operand 1 is converted to ordinal operand 2. The setting of the Rounding Control bits determines the type of rounding that is used. If the rounded value is < 0 or > 4,294,967,295, the Ordinal Overflow Fault is raised.

CONVERT TEMPORARY REAL TO INTEGER CVT_TR_I

ID#	Operands 1	2	3
170	tr	i	-

Opcode	Reference	Format	Class
01	varies	varies	110101

Temporary-real operand 1 is converted to integer operand 2. The setting of the Rounding Control bits determines the type of rounding that is used. If the rounded value is < -2,147,483,648 or > 2,147,483,647, the Integer Overflow Fault is raised.

CONVERT TEMPORARY REAL TO SHORT REAL CVT_TR_SR

ID#	Operands		
	1	2	3
171	tr	sr	-

Opcode	Reference	Format	Class
11	varies	varies	110101

Temporary-real operand 1 is converted to short-real operand 2. The setting of the Rounding Control bits determines the type of rounding that is used. The setting of the Precision Control bits has no effect. If the magnitude of the rounded value is too large to be represented as a short real, the Short Real Overflow Fault is raised. If the magnitude of the rounded value is too small to be represented as a short real, the Short Real Underflow Fault is raised.

CONVERT TEMPORARY REAL TO REAL CVT_TR_R

ID#	Operands		
	1	2	3
172	tr	r	-

Opcode	Reference	Format	Class
none	varies	varies	001101

Temporary-real operand 1 is converted to real operand 2. The setting of the Rounding Control bits determines the type of rounding that is used. The setting of the Precision Control bits has no effect. If the magnitude of the rounded value is too large to be represented as a real, the Real Overflow Fault is raised. If the magnitude of the rounded value is too small to be represented as a real, the Real Underflow Fault is raised.

OBJECT OPERATORS

SUB-OPERATOR PROCEDURES

Due to the complexity of the object operators, a more algorithmic description of each operator's action is used in the rest of this chapter. The following procedures are used throughout the descriptions of the object operators. Whenever these procedures are mentioned in the action descriptions, the procedure's action as defined below is performed. Additional sub-operator procedures are defined for specific functional groups of object operators (e.g., for the process communication operators).

Set Copied

- If the access descriptor is access-valid, set the copied bit of its associated object descriptor. Otherwise, do nothing.

Level Check

- If the specified access descriptor is access-valid, if its Unchecked Copy Rights bit is 0, and if the level number of its associated object descriptor is greater than the destination's level, then raise the Level Fault. Otherwise do nothing.

Store AD

- Perform Level Check of the specified access descriptor with that of the destination access environment. Note that if the destination access environment is specified by a refinement, the level number of the base object, rather than that of the refinement, is used in the level check.
- Read the access descriptor that is in the destination access environment location.
- If that AD is access-valid and does not have Delete Rights, raise the Destination Delete Rights Fault.
- Otherwise, store the specified access descriptor (setting Delete Rights) into the destination access descriptor location.

Object Locking

- If the lock mode in the object lock field is 00, indivisibly update the object lock with the lock mode and locker ID specified by the operation that has invoked this procedure.
- Otherwise, wait 300 machine cycles and retry the operation.
- If the object lock cannot be locked after 32 retries, return from Object Locking with a status of unsuccessful.

OD Allocation

- If stack allocation is specified by the operation that invoked this procedure, allocate an object descriptor from the current process object table.
- Otherwise, for heap allocation, allocate a Free Entry from the object table referenced by the specified SRO.
- Create an image for the AD to the newly allocated object descriptor with all access rights and with Delete Rights. If the Allocation Level = 0, then set Unchecked Copy Rights; otherwise clear it. Return the AD image to the operation that invoked this procedure.

Segment Allocation

- Initialize the newly allocated OD to a storage descriptor with the specified object type, the specified access part and data part lengths, and the appropriate level, and clear the Completed bit (in the storage descriptor) to 0 and initialize the defining TDO as the specified TDO.
- If heap allocation is specified by the operation that invoked this procedure, a rotating first-fit algorithm is used to search the physical storage object specified by the SRO for a storage block of sufficient size.
- Otherwise, for stack allocation, the single storage block in the current process physical storage object is used.
- Allocate from the selected storage block.
- Initialize the Base Address field in the new storage descriptor to the fence address in the object that is represented by the newly allocated segment.
- If the storage block is dirty (as indicated by the Dirty bit in the Storage Block Specifier) and the sum of the AP and DP rounded lengths is greater than 2,304 bytes, raise the Clear Memory Size fault. Otherwise, clear the new segment to zeros and set the Completed bit in the new segment's storage descriptor.

BRANCH OPERATORS

If a branch reference in a branch instruction specifies a displacement, either relative or absolute, to a point that is outside the boundary of the object containing the target instruction, an Instruction Object Displacement Fault occurs.

BRANCH BR

ID#	Operand 1	Operand 2	Operand 3
**	-	-	-

Opcode	Reference	Format	Class
none	bref	none	100110

Operation: A branch is made within the current instruction object to the target instruction specified by a branch reference.

BRANCH TRUE BR_T

ID#	Operand 1	Operand 2	Operand 3
**	b	-	-

Opcode	Reference	Format	Class
0	bref¦dref	varies	1000

Operation: If the boolean value specified by operand 1 is TRUE, a branch is made in the instruction stream within the current instruction object to the target instruction specified by a branch reference. The data reference for operand 1 must be encoded before the branch reference in a BR_T instruction's Reference field.

BRANCH FALSE BR_F

ID#	Operand 1	Operand 2	Operand 3
**	b	-	-

Opcode	Reference	Format	Class
1	bref¦dref	varies	1000

Operation: If the boolean value specified by operand 1 is FALSE, a branch is made in the instruction stream within the current instruction object to the target instruction specified by the branch reference. The data reference for operand 1 must be encoded before the branch reference in a BR_F instruction's Reference field.

BRANCH INDIRECT BR_INDIRECT

ID#	Operands 1	2	3
173	so	-	-

Opcode	Reference	Format	Class
0010	varies	varies	0000

Operation: Short-ordinal operand 1 is used as the new value for the instruction pointer. This causes a branch within the current instruction object to the instruction whose bit displacement from the base of the object is given by operand 1.

BRANCH INTERSEGMENT BR_ISEG

ID#	Operands 1	2	3
174	pw	-	-

Opcode	Reference	Format	Class
001	varies	varies	110110

Operation: Branches to a target instruction in a specific instruction object in the defining domain of the current context.

Operand 1: A packed word operand comprised of the following:

0 - 15: Contains the domain access index for the new instruction object.

16 - 31: Contains the short-ordinal bit displacement from the base of the new instruction object to the first bit of the instruction where execution is to continue.

Action:

- Branch to the instruction at the specified displacement into the instruction object specified by the DAI in operand 1.

BRANCH INTERSEGMENT WITHOUT TRACE BR_ISEG_WO_TRACE

ID#	Operands 1	2	3
175	pw	-	-

Opcode	Reference	Format	Class
101	varies	varies	110110

Operation: This operator operates identically to the BRANCH INTERSEGMENT operator with the exception that it is immune to all trace events at the end of this instruction.

BRANCH INTERSEGMENT AND LINK BR_ISEG_LINK

ID#	Operands 1	2	3
176	pw	pw	

Opcode	Reference	Format	Class
1011	varies	varies	1100

Operation: Branches to a target instruction in a specified instruction object in the defining domain of the current context and stores the necessary linkage information to allow later return.

Operand 1: A packed word operand comprised of the following:

0 - 15: Contains the domain access index for the new instruction object.

16 - 31: Contains the short-ordinal bit displacement from the base of the new instruction object to the first bit of the instruction where execution is to continue.

Operand 2: A packed word operand comprised of the following:

0 - 15: Is used as storage for the domain access index to the current instruction object.

16 - 31: Is used to store the short-ordinal bit displacement from the base of the current instruction object to the first bit of the instruction where execution is to continue (i.e., the current instruction pointer).

Action:

- Store the linkage information in operand 2.
- Branch to the instruction at the specified displacement into the instruction object specified by the DAI in operand 1.

BREAKPOINT BREAKPOINT

ID#	Operands 1	2	3
177	-	-	-

Opcode	Reference	Format	Class
none	none	none	111111

Operation: Branches to bit displacement 64 in the trace instruction object within the current defining domain and stores the necessary linkage information to allow later return. The appropriate linkage information is stored in the Trace DAI and Trace Instruction Pointer fields in the Trace Control Data Area of the current context. A trace code of 5 is written in the Trace Code field of this same area.

ACCESS DESCRIPTOR OPERATORS

COPY ACCESS DESCRIPTOR COPY_AD

ID#	Operands 1	2	3
178	as	as	-

Opcode	Reference	Format	Class
0001	varies	varies	0100

Operation: Copies an access descriptor from a specified location in any directly accessible environment to another specified location in any directly accessible environment.

Operand 1: Contains an access selector for the source access descriptor to be copied.

Operand 2: Contains an access selector for the destination access descriptor location.

Action:
- Perform Set Copied for the source AD.
- Perform Store AD of the source AD into the destination AD location.

NULL ACCESS DESCRIPTOR NULL_AD

ID#	Operands 1	2	3
179	as	-	-

Opcode	Reference	Format	Class
1010	varies	varies	0000

Operation: Logically clears a given AD location, overwriting the previous AD in the specified location with a null AD. Access paths using the AD in this location to reference the object are thus disconnected from the object.

Operand 1: Contains the access selector for the destination access descriptor location.

Action:
- Perform Store AD of a null access descriptor into the destination AD location.

TYPE AND RIGHTS MANIPULATION OPERATORS

AMPLIFY RIGHTS AMPLIFY_RIGHTS

ID#	Operands 1	2	3
180	as	as	-

Opcode	Reference	Format	Class
1001	varies	varies	0100

Operation: Amplifies, under control of a type control object, the selected rights bits in the specified source access descriptor.

Operand 1: Contains the access selector for a type control object. The selected AD must have Amplify Rights.

Operand 2: Contains the access selector for the access descriptor that is to be amplified. This source access descriptor is both a source and a destination.

Action:

- If the source AD is not access valid, do nothing for this entire operation. Otherwise:
- Read the contents of the type control object.
- If the Type Test bit in the type control object is 1, then:
  - If the object type of the source AD does not match that contained in the type control object, raise the Type Fault.
  - If the Dynamic/System bit is 1, the Defining TDO AD in the source OD must match the Defining TDO AD in the type control object; otherwise, raise the Type Fault.
- If the AD to be amplified is changed between type testing and amplification, perform no amplification and raise the Race Condition Fault.
- Otherwise, logically OR the Delete, Unchecked Copy, Read, Write, and Type Rights of the TCO into their corresponding fields in the source/destination AD.

RESTRICT RIGHTS

RESTRICT_RIGHTS

ID#	Operands 1	2	3
181	o	as	-

Opcode	Reference	Format	Class
011	varies	varies	101001

Operation: Restricts, under control of an ordinal bit mask, the set of rights in the specified access descriptor.

Operand 1: Contains an ordinal that has the appropriate rights bit values that are required as a mask by this instruction. The bits values are interpreted at the same bit offset that they have in an AD. The mask bit offsets are as follows:

- 1 - 3: Type Rights field
- 16: Delete Rights
- 17: Unchecked Copy Rights
- 18: Read Rights
- 19: Write Rights

Operand 2: Contains the access selector for the access descriptor to be restricted. This source AD is also the destination.

Action:

- If the source AD is not access valid or the Delete Rights bit is 0, do nothing.
- Otherwise, clear the rights bit in the source/destination AD provided that the corresponding rights bit is 1 in operand 1.

RETRIEVE TYPE DEFINITION RETRIEVE_TYPE_DEF

ID#	Operands 1	2	3
182	as	as	-

Opcode	Reference	Format	Class
0101	varies	varies	0100

Operation: Retrieves the type definition object associated with an dynamic-type or system-type object.

Operand 1: Contains the access selector for the dynamic-type or system-type object.

Operand 2: Contains the access selector for the destination access descriptor location.

Action:

- If the source AD is not access-valid, raise the Source AD Validity fault.
- If the object type of the source OD is generic, use the generic TDO AD (specified in the processor object) as the type definition AD.
- If the source OD is a storage descriptor, then use the defining TDO AD Image in the storage descriptor as the type definition AD.
- If the source OD is a refinement descriptor, then use the defining TDO AD Image in the base storage descriptor of the refinement.
- Perform Set Copied for the type definition AD.
- Perform Store AD of an image of that type defintion AD into the destination access descriptor location.

REFINEMENT OPERATORS

CREATE REFINEMENT CREATE_RFN

ID#	Operands 1	2	3
183	as	pw	pd

Opcode	Reference	Format	Class
none	varies	varies	010011

Operation: Creates a refinement of a source object given an SRO and specified offsets and lengths.

Operand 1: Contains an access selector for a storage resource object. The selected AD must have Create Rights. If this operand is zero, the current process allocation stack is used for allocation.

Operand 2: A packed word operand comprised of the following:

0 - 15: Contains the access selector for the destination access descriptor location.

16 - 31: Contains the access selector for the source object to be refined.

Operand 3: A packed doubleword operand comprised of the following:

0: Contains the bit for specifying refinement with a data part.

1 - 15: Contains a short-ordinal value which specifies the (length-1)/2 in bytes of the access part of the refined object. A value of zero specifies a null access part.

16 - 31: Contains a short-ordinal value which specifies the offset displacement in bytes from the base of the object being refined to the base of the access part of the refinement.

32 - 47: Contains a short-ordinal value which specifies the length-1 in bytes of the data part of the refined object.

48 - 63: Contains a short-ordinal value which specifies the offset displacement in bytes from the base of the object being refined to the base of the data part of the refinement.

Action:

- If the source AD is not access-valid, raise the Source AD Validity Fault.
- Perform Set Copied for the source AD.
- If the source AD does not reference an associated storage descriptor or refinement descriptor, then raise the Object Descriptor Type Fault.
- Raise the Offset and Length Compatibility Fault if:
  - the specified AP offset is not an integral multiple of 4.
  - the sum of the AP offset and AP length-1 is greater than 65,535.
  - If the DP Valid bit is 1 and the sum of DP offset and DP length-1 is greater than 65,535.

- Raise the Refinement Overflow Fault if:
  - The DP valid bit is 1, and the sum of the DP offset and DP length is greater than the DP length of the source object.
  - The sum of the AP offset and AP length is greater than the AP length of the source object.
- Perform OD Allocation.
- Perform Level Check of the source AD with the (destination) level in the new OD.
- Initialize the new OD to a refinement descriptor of generic object type.
  - The AP Length and DP Length fields are initialized to the values specified by Operand 3.
  - If the source AD references an associated storage descriptor:
    - Initialize the Base Directory Index and Base Segment Index fields to be the same as the Directory Index and Segment Index fields in the source AD.
    - Initialize the AP Offset and DP Offset fields to the values specified by Operand 3.
  - Otherwise, if the source AD references an associated refinement descriptor:
    - Initialize the Base Directory Index and Base Segment Index fields to be the same as those in the associated refinement descriptor.
    - Initialize the AP Offset field to the sum of the AP Offset of the associated refinement descriptor and the AP offset specified by Operand 3.
    - Initialize the DP Offset field to the sum of the DP Offset of the associated refinement descriptor and the DP offset specified by Operand 3.
- Perform Store AD of the AD image for the new refined object into the destination access descriptor location as follows:
  - Make the Read Rights and Write Rights the same as the corresponding rights in the source AD.
  - Write the Type Rights field to match the corresponding rights in the source AD.

CREATE TYPED REFINEMENT CREATE_TYPED_RFN

ID#	Operands 1	2	3
184	pd	pd	-

Opcode	Reference	Format	Class
11	varies	varies	000101

Operation: Creates a typed refinement of the source object given an TCO, an SRO, and specified offsets and lengths.

Operand 1: A packed doubleword operand comprised of the following:

0 - 15: Contains the access selector for a type control object. The selected ad must have Refine Rights.

16 - 31: Contains the access selector for a storage resource object. The selected AD must have Create Rights. If this operand is zero, the current process allocation stack is used for allocation.

32 - 47: Contains the access selector for the destination access descriptor location.

48 - 63: Contains the access selector for the source object to be refined

Operand 2: A packed doubleword operand comprised of the following:

0: Contains the bit for specifying refinement with a data part.

1 - 15: Contains a short-ordinal value which specifies the (length-1)/2 in bytes of the access part of the refined object. A value of zero specifies a null access part.

16 - 31: Contains a short-ordinal value which specifies the offset displacement in bytes from the base of the object being refined to the base of the access part of the refinement.

32 - 47: Contains a short-ordinal value which specifies the length-1 in bytes of the data part of the refined object.

48 - 63: Contains a short-ordinal value which specifies the offset displacement in bytes from the base of the object being refined to the base of the data part of the refinement.

Action:

- If the source AD is not access valid, raise the Source AD Validity Fault.
- Perform Set Copied for the source AD.
- If the source AD does not reference an associated storage descriptor or refinement descriptor, then raise the Object Descriptor Type Fault.
- If the source object type does not match the object type in the TCO, then raise the Type Fault.
- Raise the Offset and Length Compatibility Fault if:
  - the specified AP offset is not an integral multiple of 4.
  - the sum of the AP offset and AP length-1 is greater than 65,535.
  - If the DP valid bit is 1 and the sum of DP offset and DP length-1 is greater than 65,535.

- Raise the Refinement Overflow Fault if:
  - The DP valid bit is 1 and the sum of the DP offset and DP length is greater than the DP length of the source object.
  - The sum of the AP offset and AP length is greater than the AP length of the source object.
- Perform OD Allocation.
- Perform Level Check of the source AD with the (destination) level in the new OD.
- Initialize the new OD to a refinement descriptor of the object type specified by the Object Type field in the TCO:
  - Initialize the AP Length and DP Length fields to that specified by Operand 2.
  - If the source AD references an associated storage descriptor:
    - Initialize the Base Directory Index and Base Segment Index fields to be the same as the Directory Index and Segment Index fields in the source AD.
    - Initialize the AP Offset and DP Offset fields to the values specified by Operand 2.
  - Otherwise, if the source AD references an associated refinement descriptor:
    - Initialize the Base Directory Index and Base Segment Index fields to be the same as those in the associated refinement descriptor.
    - Initialize the AP Offset field to the sum of the AP Offset of the associated refinement descriptor and the AP offset specified by Operand 2.
    - Initialize the DP Offset field to the sum of the DP Offset of the associated refinement descriptor and the DP offset specified by Operand 2.
- Perform Store AD of the AD image for the new refined object into the destination access descriptor location as follows:
  - Make the Read and Write Rights fields the same as the corresponding rights in the source AD.
  - Write the Type Rights field to match the corresponding rights in the source AD.

OBJECT CREATION OPERATORS

CREATE OBJECT CREATE_OBJ

ID#	Operands 1	Operands 2	Operands 3
185	as	as	pw

Opcode	Reference	Format	Class
0	varies	varies	000011

Operation: Creates an object with the specified access and data part lengths and an access descriptor (with associated OD) for the new object.

Operand 1: Contains the access selector for a storage resource object. The selected AD must have Create Rights. If this operand is zero, the current process allocation stack is used for allocation.

Operand 2: Contains the access selector for the destination access descriptor location.

Operand 3: A packed word operand comprised of the following:

0 - 15: Contains the short-ordinal length-1 of the data part of the object to be created.

16: Contains the boolean which specifies whether the object has a data part or not.

17 - 31: Contains the (length-1)/2 of the access part of the object to be created. A value of zero specifies a null access object part in the new object.

Action:

- Perform OD Allocation.
- Perform Segment Allocation for a generic object using a null AD as the TDO-AD Image.
- Perform Store AD of the AD for the new storage descriptor into the specified destination access descriptor location.

CREATE TYPED OBJECT CREATE_TYPED_OBJ

ID#	Operands 1	2	3
186	as	pw	pw

Opcode	Reference	Format	Class
01	varies	varies	100011

Operation: Creates a typed object with specified access and data part lengths and an access descriptor (with associated OD) for the new object.

Operand 1: Contains the access selector for a storage resource object. The selected AD must have Create Rights. If this operand is zero, the current process allocation stack is used for allocation.

Operand 2: A packed word operand comprised of the following:

0 - 15: Contains the access selector for a type control object. The selected AD must have Create Rights.

16 - 31: Contains the access selector for the destination access descriptor location.

Operand 3: A packed word operand comprised of the following:

0 - 15: Contains the short-ordinal length-1 of the data part of the object to be created.

16: Contains the bit which specifies whether the object has a data part or not.

17 - 31: Contains the (length-1)/2 of the access part of the object to be created. A value of zero specifies a null access part.

Action:

- Perform Set Copied for the Defining TDO AD in the TCO.
- Perform OD Allocation.
- Perform Level Check of the Defining TDO AD in the TCO with the (destination) level in the new OD.
- Perform Segment Allocation for an object with Object Type the same as specified in the TCO and use the Defining TDO AD in the TCO as the defining TDO AD Image in the newly allocated storage descriptor.
- Preform Store AD of the AD for the new object into the specified destination access descriptor location.

ACCESS INSPECTION OPERATORS

INSPECT ACCESS DESCRIPTOR INSPECT_AD

ID#	Operands 1	2	3
187	as	o	-

Opcode	Reference	Format	Class
110	varies	varies	010001

Operation: Copies the image of an access descriptor into an ordinal in the data part of an object.

Operand 1: Contains the access selector for the source access descriptor that is to be inspected.

Operand 2: Is an ordinal destination for the image of the source access descriptor.

Action: • Read the source access descriptor and write an image of it into ordinal operand 2.

INSPECT OBJECT INSPECT_OBJ

ID#	Operands 1	2	3
188	as	pw	-

Opcode	Reference	Format	Class
001	varies	varies	010001

Operation: Copies an image of an access descriptor and its associated object descriptor into the data part of an object beginning at a specified location.

Operand 1: Contains the access selector for the source access descriptor that is to be inspected.

Operand 2: A packed word operand comprised of the following:

0 - 15: Contains the access selector for the destination object.

16 - 31: Contains a short-ordinal which specifies the byte displacement within the data part of the destination object to the location where the inspection data is to be stored.

Action: • Write a 20-byte record beginning at the specified location in the data part of the destination object. The record contains the following:

- An image of the source AD in the first 4 bytes.
- An image of the associated OD of the source AD in the last 16 bytes.

EQUAL ACCESS EQL_ACCESS

ID#	Operands 1	2	3
189	as	as	b

Opcode	Reference	Format	Class
0011	varies	varies	111101

Operation: Compares two source access descriptors for equality.

Action:
- If the two source access descriptors are access-valid and their Directory and Segment indices are equal or both are not access-valid, return a boolean value of TRUE to the destination. Otherwise, return a boolean FALSE.

MOVE TO EMBEDDED DATA VALUE MOV_TO_EDV

ID#	Operands 1	2	3
190	o	as	-

Opcode	Reference	Format	Class
111	varies	varies	101001

Operation: Copies an ordinal value into an embedded data value in a destination access descriptor location.

Operand 1: Contains an ordinal value to be copied to the destination AD location.

Operand 2: Contains the access selector for the destination AD location.

Action:
- Compose an embedded data value using the most-significant 31 bits of operand 1 (bit 0 is cleared).
- Perform Store AD of the embedded data value into the destination AD location specified by operand 2.

MOVE FROM EMBEDDED DATA VALUE MOV_FM_EDV

ID#	Operands 1	2	3
187	as	o	-

Opcode	Reference	Format	Class
110	varies	varies	010001

Operation: Copies the image of an embedded data value into an ordinal in the data part of an object. This operator is identical to the INSPECT ACCESS DESCRIPTOR operator.

Operand 1: Contains the access selector for the source AD location.

Operand 2: Contains the ordinal destination for the embedded data value.

Action:
- Read the source access descriptor location specified by operand 1 and write an image of that AD into operand 2. The least-significant bit of operand 2 will be zero if the result is an embedded data value.

ACCESS INTERLOCK OPERATORS

LOCK OBJECT LOCK_OBJ

ID#	Operands 1	2	3
191	as	so	b

Opcode	Reference	Format	Class
1011	varies	varies	111101

Operation: Locks an object lock at a specified location within the data part of an object.

Operand 1: Contains the access selector for the object that contains the object lock.

Operand 2: Contains a short-ordinal byte displacement within the data part of the selected object to the object lock field.

Operand 3: Contains a boolean result that is set TRUE if the lock operation is successful.

Action:

- Perform Object Locking at the specified location using a Long-Term Software Lock.
- If the lock operation is successful, the boolean result is TRUE. Otherwise, the result is FALSE. The boolean result is stored in destination operand 3.

UNLOCK OBJECT UNLOCK_OBJ

ID#	Operands 1	2	3
192	as	so	-

Opcode	Reference	Format	Class
1101	varies	varies	0100

Operation: Unlocks an object lock at a specified location within a data part of an object.

Operand 1: Contains the access selector for the object that contains the object lock.

Operand 2: Contains a short-ordinal byte displacement within the data part of the selected object to the object lock field.

Action:

- If the Lock Mode field in the specified object lock is 10 (i.e., long-term software locked), and the Locker ID field is equal to the current Process ID (reflected in the current process object), the Object Lock field is indivisibly cleared to zeros.
- Otherwise, raise the Object Lock ID/Type Fault.

INDIVISIBLY ADD SHORT ORDINAL INDIV_ADD_SO

ID#	Operand 1	Operand 2	Operand 3
193	so	so	-

Opcode	Reference	Format	Class
0011	varies	varies	0100

Operation: Short-ordinal operand 1 is indivisibly (within one read-modify-write cycle) added to short-ordinal operand 2. The result is stored into operand 2. The original value of operand 2 is pushed onto the operand stack. A Short-Ordinal Overflow Fault cannot occur. Thus, this operator can be used to indivisibly subtract operand 1 from operand 2 if operand 1 contains the 2's complement of the number to be subtracted.

INDIVISIBLY ADD ORDINAL INDIV_ADD_O

ID#	Operand 1	Operand 2	Operand 3
194	o	o	-

Opcode	Reference	Format	Class
0111	varies	varies	1100

Operation: Ordinal operand 1 is indivisibly (within one read-modify-write cycle) added to ordinal operand 2. The result is stored into operand 2. The original value of operand 2 is pushed onto the operand stack. An Ordinal Overflow Fault cannot occur. Thus, this operator can be used to indivisibly subtract operand 1 from operand 2 if operand 1 contains the 2's complement of the number to be subtracted.

INDIVISIBLY INSERT SHORT ORDINAL — INDIV_INS_SO

ID#	Operands 1	2	3
195	bfs	so	so

Opcode	Reference	Format	Class
10111	varies	varies	0010

Operation: Indivisibly inserts a bit field from one short ordinal into another short ordinal.

Operand 1: Contains the bit-field specifier for the destination bit field that is to be written.

Operand 2: Contains a short ordinal with a right-justified bit field that is to be inserted into the destination short ordinal.

Operand 3: Contains the destination short ordinal into which the bit field is to be inserted.

Action:

- Indivisibly (within one read-modify-write cycle) insert the source field from operand 2 into the specified destination bit-field in operand 3.
- Push the original value of operand 3 onto the operand stack.

INDIVISIBLY INSERT ORDINAL — INDIV_INS_O

ID#	Operands 1	2	3
196	bfs	o	o

Opcode	Reference	Format	Class
011	varies	varies	100011

Operation: Indivisibly inserts a bit field from one ordinal into another ordinal.

Operand 1: Contains the bit-field specifier for the destination bit field that is to be written.

Operand 2: Contains an ordinal with a right-justified bit field that is to be inserted into the destination ordinal.

Operand 3: Contains the destination ordinal into which the bit field is to be inserted.

Action:

- Indivisibly (within one read-modify-write cycle) insert the source field from operand 2 into the specified destination bit-field in operand 3.
- Push the original value of operand 3 onto the operand stack.

CONTEXT OPERATORS

The following sub-operator procedures (ENV Entry and Context Call) are used in the action descriptions of context operators.

ENV Entry

- If the source AD is access-valid, then do the following:
  - Perform Set Copied for this source AD.
  - Read the level number of the object referenced by the source AD. If the source object is specified by a refinement, the level number of the base object rather than that of the refinement is read. The level number is located in the OD of the object.
- Otherwise, if the source AD is not access valid, then use the maximum level number (65,535).
- Write the level number in the appropriate Entered ENV Level field in the process object.
- Write the source AD without Delete Rights into the appropriate Access Environment AD location in the current context object.
- Save the new access environment information within the GDP.

Context Call

- If the static link access selector (specified as an operand) is 4, a null AD is used as the static link AD. Otherwise, the static link access selector is interpreted as selecting the AD to be used as the static link.
- Perform Set Copied for the AD of the specified domain.
- If the domain is a refinement, then do the following:
  - Traverse to the base object by using the Base Segment and Base Directory indices (in the refinement descriptor).
  - Adjust the called instruction object's DAI to that relative to the base domain by adding the AP Offset in the refinement descriptor.
- Read the AD of the called instruction object using the adjusted domain access index.
- If the called instruction object AD has no Call Rights, raise the Instruction Object Type Rights Fault.
- Read the instruction object header (the first 8 bytes of the instruction object).
- If either the Context Data Part Length or the Context Access Part Length (from the instruction object header) is less than its respective minimum size, raise the Context Parameters Size Fault.
- If either the Context Data Part Length or the Context Access Part Length (from the instruction object header) is greater than its respective current size (of the pre-created context) in the process object, raise the Context Parameters Size Fault.
- Increment the Current Allocation Level in the process object by 1.

- Read the Context Link AD in the current context.
- Update the AP Length and DP Length of the new context to that specified in the instruction object header.
- Initialize the context access part starting with the AD 14 location to null ADs. Initialize the data part to zeros.
- Initialize the new context access part:
  - Write into the Defining Domain location an AD (with Write Rights and without Delete Rights) for the specified domain (the base domain after any refinement traversal).
  - Write into the Local Constants location an AD for the object specified by the Local Constants DAI field in the called instruction object's header.
  - Write into AD location 5 (Environment 1) the AD for the defining domain.
  - Write null ADs into AD locations 6, and 7 for the initial environments 2 and 3.
  - Write the Top of Descriptor Stack AD and Top of Storage Stack AD of the current context into the corresponding locations.
  - Write into Static Link location the specified static link AD.
- Initialize the new context data part as follows:
  - Copy the current context status into the Context Status field.
  - Write into the Operand Stack Pointer field the value from the Initial Operand Stack Pointer field in the called instruction object's header.
  - Write into the Current Instruction Object DAI field the adjusted domain access index of the instruction object (adjusted above only in the case of refinement).
  - Write into the Instruction Pointer field an initial value of 64.
- Set up the return information for the current (calling) context as follows:
  - Write the current values into the following current context data part:
    - Context Status
    - Operand Stack Pointer
    - Current Instruction Object DAI
    - Instruction Pointer

    Write the Instruction Pointer with the value pointing to the next instruction to be executed upon return from the called context.
- Write an AD for the new context into the Current Context location in the process object.
- Initialize the Entered ENV 1 Level field in the current process object with the level number of the defining domain.
- Initialize the Entered ENV 2 and 3 Level fields in the current process object to maximum.
- Replace the GDP's internal context environment with that of the called context and continue execution at the instruction specified.

ENTER ENVIRONMENT 1 ENTER_ENV_1

ID#	Operands 1	2	3
197	as	-	-

Opcode	Reference	Format	Class
0110	varies	varies	0000

Operation: Changes environment 1 of the current context to allow direct access to the access descriptors in a specified object.

Operand 1: Contains the access selector for the object to be entered.

Action: • Perform ENV Entry of the source AD into AE 1.

ENTER ENVIRONMENT 2 ENTER_ENV_2

ID#	Operands 1	2	3
198	as	-	-

Opcode	Reference	Format	Class
1110	varies	varies	0000

Operation: Changes environment 2 of the current context to allow direct access to the access descriptors in a specified object.

Operand 1: Contains the access selector for the object to be entered.

Action: • Perform ENV Entry of the source AD into AE 2.

ENTER ENVIRONMENT 3 ENTER_ENV_3

ID#	Operands 1	2	3
199	as	-	-

Opcode	Reference	Format	Class
0001	varies	varies	0000

Operation: Changes environment 3 of the current context to allow direct access to the access descriptors in a specified object.

Operand 1: Contains the access selector for the object to be entered.

Action: • Perform ENV Entry of the source AD into AE 3.

COPY PROCESS GLOBALS COPY_PRCS_GLOBALS

ID#	Operands 1	2	3
200	as	-	-

Opcode	Reference	Format	Class
1001	varies	varies	0000

Operation: Copies an access descriptor for the current process's process globals object into the specified AD location.

Operand 1: Contains an access selector for the destination access descriptor location.

Action:
- Read the Process Globals AD (in the current process object).
- Perform Set Copied for this AD.
- Perform Store AD of this AD into the destination AD location.

SET CONTEXT MODE SET_CTXT_MODE

ID#	Operands 1	2	3
201	so	-	-

Opcode	Reference	Format	Class
0101	varies	varies	0000

Operation: Writes the value of short-ordinal operand 1 to the Context Status field of the current context object. The context mode within the GDP is also updated accordingly.

ADJUST STACK POINTER ADJ_SP

ID#	Operands 1	2	3
202	si	so	-

Opcode	Reference	Format	Class
01011	varies	varies	0100

Operation: Adds short-integer operand 1 to the operand stack pointer, returning the previous stack pointer value to short-ordinal destination operand 2.

Action:
- Save the current operand stack pointer temporarily. The value saved is after operand and address evaluation.
- Add short-integer operand 1 to the current operand stack pointer. A short-integer overflow fault cannot occur.
- Store the saved operand stack pointer value into operand 2. If the destination is the operand stack, the destination is specified by the new stack pointer.

CALL CALL

ID#	Operands 1	2	3
203	as	pw	-

Opcode	Reference	Format	Class
101	varies	varies	010001

Operation: Creates a new context using an instruction object in a directly accessible domain and then calls that new context while passing a static link to it.

Operand 1: Contains the access selector for the static link of the new context.

Operand 2: A packed word operand comprised of the following:

0 - 15: Contains the access selector for the defining domain of the new context.

16 - 31: Contains the domain access index into the specified defining domain of an access descriptor for the instruction object for which the context is to be created.

Action:

- Perform Context Call of the specified instruction object using the static link AD specified by operand 1.

CALL THROUGH DOMAIN CALL_THRU_DOMAIN

ID#	Operands 1	2	3
204	as	pw	-

Opcode	Reference	Format	Class
011	varies	varies	010001

Operation: Creates a new context using an instruction object in a specified new defining domain that is within the current defining domain and then calls that new context while passing a static link to it.

Operand 1: Contains the access selector for the static link to be passed.

Operand 2: A packed word operand comprised of the following:

0 - 15: Contains the domain access index for an access descriptor for the defining domain of the new context.

16 - 31: Contains the domain access index into the specified defining domain of an access descriptor for the instruction object for which the context is to be created.

Action:

- Perform Context Call of the specified instruction object using the static link AD specified by operand 1.
- Enter the new domain as Environment 1.

RETURN RET

ID#	Operands 1	2	3
205	-	-	-

Opcode	Reference	Format	Class
0	none	none	000110

Operation: Returns from the currently active context to the calling context.

Action:

- If tracing is enabled and the Trace Mode field (in the Process Status field) specifies Flow Trace, then raise the Trace Event Fault and thus continue execution in the Trace Instruction Object (referenced by AD 1 in the domain access part).
- If the Calling Context AD does not have Return Rights, raise the Context Type Rights Fault.
- If the Top of Storage Stack AD of the current context differs from that of the calling context, use the Top of Storage Stack AD (in the calling context) to calculate an end address to update (and set the Dirty bit of) the allocation stack specifier in the process's physical storage object. This deallocates the local storage allocated for this returning context.
- If the Top of Descriptor Stack AD (in the current context) differs from that of the calling context, use the Segment Index of the Top of Descriptor Stack AD (in the calling context) to update the Free Index field of the Header Entry in the process object table. This deallocates all ODs in the process object table that were allocated to this returning context.
- Decrement by 1 the Current Allocation Level field of the process object.
- Write the AD from the Calling Context location (in the returning context object) into the Current Context location in the process object. That is, make the previous calling context the current context.
- Replace the GDP's internal context environment with that of the calling context and continue execution at the instruction specified in the Instruction Pointer field of the new current context object.

RETURN AND FAULT RET_FAULT

ID#	Operands 1	2	3
206	-	-	-

Opcode	Reference	Format	Class
1	none	none	000110

Operation: Returns from the currently active context and resumes execution at bit displacement 64 in the Fault Instruction Object specified in the defining domain of the context returned to.

Action:

- Perform the RETURN operator.
- Raise the Return Fault.

PROCESS COMMUNICATION OPERATORS

The following sub-operator procedures (Enqueue Message, Dequeue Message, Enqueue Carrier, Dequeue Carrier, Forward Carrier, Surrogate Common, Send Common, and Receive Common) are used in the action descriptions of process communication operators.

Enqueue Message

- Remove a port message queue entry from the free list.
- Write the specified message AD into the AD location (in the Message Queue Access Area) corresponding to the new message queue entry.
- If the Queue Discipline (in the Port Status) is FIFO, insert the new entry into the message queue at the tail of the message queue linked list.
- Otherwise, search the message queue linked list and, depending on the queuing values in the message queue entries, find an appropriate insertion point. Insert the new entry into the message queue at this point with the appropriate queuing value.

Dequeue Message

- If the Port Type is Delay (in the Port Status field) and the Deadline value in the head entry of the message queue is positive, then return from Dequeue Message.
- Otherwise, because the Port Type is not Delay, remove the head entry of the message queue and return its corresponding message AD (from the Port Message Access Area). The AD location (from which this dequeued message AD was obtained) is written with a null AD, and the dequeued entry is returned to the free list.

Enqueue Carrier

- Enqueue the specified carrier at the tail of the carrier queue.

Dequeue Carrier

- Dequeue a carrier from the head of the carrier queue.

Forward Carrier

- If the AD for the Second Port in the current Carrier is null, then unlock the carrier and return from Forward Carrier. Otherwise, continue.
- Write that AD for the Second Port into the Current Port location in the current Carrier. That is, make the second port the current port.
- Write a null AD in the second port location of the current carrier to prevent forwarding the second time.
- Use the AD for the Second Message in the current carrier as the specified message AD.
- Use the Second Port Queuing Value in the current carrier as the current queuing value.
- Perform Send Common.

Surrogate Common

- If the specified carrier is a refinement, traverse the refinement by using the Base Segment and Base Directory indices in the refinement descriptor.
- Use the unrefined (base object) carrier as both the current carrier and the current surrogate carrier.
- Perform Object Locking on the current carrier. If locking fails, raise the Carrier Lock Fault.
- If the specified AD for the Destination Port is null, then set the Null Surrogate Destination bit, and clear the First Port Done bit in the Process Status field of the current process.
- Otherwise, clear the Null Surrogate Destination and First Port Done bits in the Process Status field of the current process.
- Write the specified AD for the destination port into the Second Port location in the current Carrier.
- If the levels of the current port, destination port, and carrier are not equal, raise the Level Fault.

Send Common

- Perform Object Locking on the current port. If the locking operation is not successful, raise the Port Lock Fault.
- If the message queue is not full and no carrier is blocked at the current port awaiting a message, then do the following:
  - Perform Enqueue Message of the specified message into the port message queue.
  - If the First Port Done or Null Surrogate Destination bit is set (in the process status), unlock the current carrier.
  - Set the First Port Done bit in the Process Status field.
  - Unlock the current port.
  - Return from Send Common.
- If the message queue is empty (i.e., the Head of Message Queue field is zero) and a carrier is blocked at the current port awaiting a message, then do the following:
  - If either the First Port Done or Null Surrogate Destination bit is set, unlock the current carrier.
  - Perform Dequeue Carrier of a carrier from the current port.
  - Make the dequeued carrier the current carrier.
  - Write the specified message AD into the Incoming Message location in the new current Carrier.
  - Set the Message Received and Unblocked bits in the Carrier Status field of the current Carrier.
  - Set the First Port Done bit in the Process Status.
  - Unlock the current port.
  - If the current carrier is a processor carrier (as indicated by the Carried Object Type field in the Carrier Status), send a Wake-Up IPC to the carried processor.
  - Otherwise, perform Forward Carrier of the carrier to its second port.
  - Return from Send Common.
- If the message queue is full (as indicated by a value of zero in the Head of Free Entry List field), then do the following:
  - If this is not a conditional SEND operation, then do the following:
    - Write the specified message AD in the Blocked Message location in the current carrier.
    - Write the current queuing value into the Blocked Queuing Value field of the current Carrier.
    - Perform Enqueue Carrier of the current carrier.
    - Unlock the current port.
    - Return from Send Common.
  - Otherwise, do the following steps for a conditional SEND operation:
    - Unlock the current port.
    - Return from Send Common with a status of unsuccessful.

Receive Common

- Perform Object Locking on the current port. If the locking operation is not successful, raise the Port Lock Fault.
- If the message queue is not empty (as indicated by a non-zero value in the Head of Message Queue field), and there is no blocked carrier awaiting, then do the following:
  - Perform Dequeue Message to obtain a message from the message queue.
  - Write the received message AD into the Incomming Message location in the current Carrier, and set the Message Received and the Unblocked bits in the Carrier Status field.
  - Set the First Port Done bit in the Process Status field.
  - Unlock the current port.
  - Return from Receive Common.
- If the message queue is full and there is a blocked carrier awaiting, then do the following:
  - Perform Dequeue Message to obtain a message from the message queue.
  - Write the received message AD into the Incoming Message location in the current Carrier and set the Message Received and Unblocked bits in the current Carrier Status field.
  - Perform Dequeue Carrier on the current port to obtain a carrier from the carrier queue.
  - Make the dequeued carrier the current carrier.
  - Perform Enqueue Message of the Blocked Message in the current carrier by using the Blocked Queuing Value field of the current carrier.
  - Write a null AD into the Blocked Message location.
  - Set the First Port Done bit in the Process Status.
  - Unlock the current port.
  - Perform Forward Carrier of the current carrier to its Second Port.
  - Return from Receive Common.
- If the message queue is empty, then do the following:
  - If this is not a conditional RECEIVE operation, then do the following:
    - Clear the Message Received and Unblocked bits in the current Carrier Status.
    - Perform Enqueue Carrier of the current carrier at the current port.
    - Unlock the current port.
    - Return from Receive Common.
  - Otherwise, do the following steps for a conditional RECEIVE operation:
    - Unlock the current port.
    - Return from Receive Common with a status of unsuccessful.

SEND SEND

ID#	Operands 1	2	3
207	as	as	-

Opcode	Reference	Format	Class
11011	varies	varies	0100

Operation: Sends a specified message to a specified port.

Operand 1: Contains the access selector for a port to which the message is to be sent. The selected AD must have Send Rights.

Operand 2: Contains the access selector for the object to be sent as a message.

Action:

- Use the Current Process Carrier as the current carrier.
- Use the port specified by operand 1 as the current port.
- Use the message AD specified by operand 2 as the message AD in further operations.
- Use a Queuing Value of zero (i.e., 0 for Priority and 0 for Deadline).
- Clear the First Port Done bit and the Null Surrogate Destination bit (in the current process status).
- If the level of the message object is less than that of the port, raise Level Fault.
- Perform Send Common.
- If the send operation is successful (as indicated by the First Port Done bit being 1 in the Process Status), continue normal execution.
- Otherwise, suspend the current process and relinquish the processor for redispatching.

RECEIVE RECEIVE

ID#	Operands 1	2	3
208	as	-	-

Opcode	Reference	Format	Class
1101	varies	varies	0000

Operation: Receives a message at a specified port.

Operand 1: Contains the access selector for the port at which the process is to receive a message. The selected AD must have Receive Rights.

Action:

- Use the Current Process Carrier as the current carrier.
- Use the port specified by operand 1 as the current port.
- Clear the First Port Done bit and the Null Surrogate Destination bit (in the current process status).
- Perform Receive Common.
- If the receive operation is successful (as indicated by the First Port Done bit being 1 in the Process Status), then write the AD of the received message into the Interprocess Message AD location in the current context and continue normal execution.
- Otherwise, set the Waiting for Message bit in the Process Status field, suspend the current process, and relinquish the processor for redispatching.

CONDITIONAL SEND COND_SEND

ID#	Operands 1	2	3
209	as	as	b

Opcode	Reference	Format	Class
0111	varies	varies	111101

Operation: Checks for the availability of message queue space at a specified port and indivisibly sends a specified message if space is available.

Operand 1: Contains the access selector for a port to which the message is to be sent. The selected AD must have Send Rights.

Operand 2: Contains the access selector for the object to be sent as a message.

Operand 3: Contains a boolean (in the low-order byte) that is set to TRUE if the SEND operation is successful and to FALSE otherwise. The high-order byte is not affected.

Action:

- Use the Current Process Carrier as the current carrier.
- Use the port specified by operand 1 as the current port.
- Use the message AD specified by operand 2 as the message AD in further operations.
- Use a Queuing Value of zero (i.e., 0 for Priority and 0 for Deadline).
- Clear the First Port Done bit and the Null Surrogate Destination bit (in the current process status).
- If the level of the message AD is less than that of the port, raise Level Fault.
- Perform Send Common.
- If the send operation is successful (as indicated by the First Port Done bit being 1 in the Process Status), then write a result of TRUE in the destination boolean operand 3 and continue normal execution.
- Otherwise, write a result of FALSE in the destination boolean operand 3 and continue normal execution.

CONDITIONAL RECEIVE COND_RECEIVE

ID#	Operands 1	2	3
210	as	b	-

Opcode	Reference	Format	Class
111	varies	varies	100001

Operation: Checks for the availability of a message at a specified port and indivisibly receives the message if it is available.

Operand 1: Contains the access selector for the port at which the message is to be received. The selected AD must have Receive Rights.

Operand 2: Contains a boolean in the low-order byte that is set to TRUE if the RECEIVE operation is successful and to FALSE otherwise. The high-order byte is not affected.

Action:

- Use the Current Process Carrier as the current carrier.
- Use the port specified by operand 1 as the current port.
- Clear the First Port Done bit and the Null Surrogate Destination bit (in the current process status).
- Perform Receive Common.
- If the receive operation is successful (as indicated by the First Port Done bit being 1 in the Process Status), then write the AD of the received message into the Interprocess Message AD location in the current context, write a result of TRUE in the destination boolean operand 3, and continue normal execution.
- Otherwise, write a result of FALSE in the destination boolean operand 3 and continue normal execution.

SURROGATE SEND SUR_SEND

ID#	Operands 1	2	3
211	as	as	pw

Opcode	Reference	Format	Class
1	varies	varies	000011

Operation: Sends a specified message to a specified port via a specified surrogate carrier.

Operand 1: Contains the access selector for a port to which the message is to be sent. The selected AD must have Send Rights.

Operand 2: Contains the access selector for the object to be sent as a message.

Operand 3: A packed word operand comprised of the following:

0 - 15: Contains the access selector for the second port to which the surrogate carrier can be forwarded. The selected AD must have Send Rights.

16 - 31: Contains the access selector for the surrogate carrier. The selected AD must have Surrogate Rights.

Action:

- Perform Surrogate Common.
- Use the message AD specified by operand 2 as the message AD in current operations.
- Use the Second Queuing Value in the current carrier as the current queuing value.
- If the level of the message AD is less than that of the port, raise the Level Fault.
- Perform Send Common.
- If the First Port Done bit is 1, and the Null Surrogate Destination bit is 0 (i.e., the send operation did not block and the current surrogate carrier needs to be forwarded to its second port), then perform Forward Carrier of the current surrogate carrier to its second port.

SURROGATE RECEIVE SUR_RECEIVE

ID#	Operands 1	2	3
212	as	pw	-

Opcode	Reference	Format	Class
111	varies	varies	010001

Operation: The current process uses a specified surrogate carrier at a specified port to wait for a message.

Operand 1: Contains the access selector for the port at which to receive a message. The selected AD must have Recieve Rights.

Operand 2: A packed word operand comprised of the following:

0 - 15: Contains the access selector for the second port to which the surrogate carrier will be forwarded. The selected AD must have Send Rights.

16 - 31: Contains the access selector for the surrogate carrier. The selected AD must have Surrogate Rights.

Action:

- Perform Surrogate Common.
- Perform Receive Common.
- If a message has been received successfully (i.e., the First Port Done bit is 1 and the receive operation did not block), then perform Forward Carrier of the surrogate carrier to its second port.

DELAY PROCESS

DELAY_PRCS

ID#	Operands		
	1	2	3
213	si	-	-

Opcode	Reference	Format	Class
0011	varies	varies	0000

Operation: Delays the current process for a specified period of time in the normal dispatching mode. Otherwise, it reschedules the process.

Operand 1: Contains the short-integer delay period that is an appropriate delay value (in system time units) based on the resolution of the system clock. This value must be positive.

Action:

- Use the current process carrier as the current carrier.
- If the current dispatching mode (in the processor status) is Normal, do the following:
  - Write the Second Port AD into the Save Port AD in the current carrier.
  - Write a null AD into the Second Port AD location.
  - Use the Delay Port (referenced in the processor object) as the current port.
  - Use operand 1 as the Deadline value.
- If the current dispatching mode (in the processor status) is not Normal, do the following:
  - Use the Second Port (in the current carrier) as the current port.
  - Write a null AD into the Second Port Ad location in the current carrier.
  - Use the Second Port Queuing Value (in the current carrier) as the current queuing value.
- Use the Second Message (specified in the current carrier object) as the message.
- Perform Send Common.
- Suspend the current process and relinquish the processor for redispatching.

SEND PROCESS — SEND_PRCS

ID#	Operand 1	Operand 2	Operand 3
214	as	-	-

Opcode	Reference	Format	Class
1011	varies	varies	0000

Operation: Sends the current process to the specified port.

Operand 1: Contains the access selector for the port at which the current process is forwarded as a message. The selected AD must have Send Process Rights.

Action:

- Use the current process carrier as the current carrier.
- Use the port specified by operand 1 as the current port.
- Write a null AD into the Second Port AD location.
- Set the First Port Done and Null Surrogate Destination bits to 1 in the current process status.
- Use the Second Port Queuing Value in the current carrier as the current queuing value.
- Use the Second Message (specified in the current carrier object) as the message.
- Perform Send Common.
- Suspend the current process and relinquish the processor for redispatching.

SET PROCESS MODE — SET_PRCS_MODE

ID#	Operand 1	Operand 2	Operand 3
215	as	so	-

Opcode	Reference	Format	Class
00111	varies	varies	0100

Operation: Updates the process status of a specified process.

Operand 1: Contains an access selector for an AD for the current process object. The selected AD must have Set Process Mode Rights.

Operand 2: Contains the short-ordinal for the new process status.

Action:

- If the source AD is not equal to the current process object AD, raise the Process Object Access Mismatch fault.
- Store the new process status (while preserving the old values of the Unbounded bit, the Process Faulted bit, and the Trace Enable bit) into the Process Status field of the process object. Update the process status within the GDP accordingly.

READ PROCESS CLOCK READ_PRCS_CLOCK

ID#	Operands 1	2	3
216	o	-	-

Opcode	Reference	Format	Class
011	varies	varies	110110

Operation: The 32-bit process clock value is read from the Process Clock field in the current process data part, updated to include the time (in system time units) consumed during the current service period, and stored in destination ordinal operand 1.

PROCESSOR COMMUNICATION OPERATORS

SEND TO PROCESSOR SEND_PSOR

ID#	Operands 1	2	3
217	as	so	b

Opcode	Reference	Format	Class
1111	varies	varies	111101

Operation: Sends an interprocessor message to a specified processor (possibly the one this instruction is executing on) via the interprocessor communication (IPC) mechanism.

Operand 1: Contains the access selector for the destination PCO. The selected AD must have Send IPC Rights.

Operand 2: Contains the short-ordinal value for the IPC message code.

Operand 3: Contains a boolean that is set to TRUE if the IPC is successfully sent and otherwise, is set to FALSE.

Action:

- Perform Object Locking on the PCO specified by operand 1.
- If the locking operation is not successful, then store a boolean result of FALSE into destination operand 3 and end this instruction.
- If the locking operation is successful and the Response Count field is not zero, then unlock the PCO and store a boolean FALSE in the destination operand 3, and end this instruction.
- If the locking operation is successful, then do the following:
  - In the PCO, set the Response Count field to the value of Processor Count. This field will be decremented by the receiving processor.
  - Copy the IPC message code specified by operand 2 into the IPC Message Code field of the PCO.
  - Signal an IPC by writing into IPC register address 2 in the interconnect address space the value of the Processor ID from the specified PCO.
  - Return boolean TRUE to the destination operand 3.
  - Unlock the PCO.

READ PROCESSOR STATUS READ_PSOR_STATUS

ID#	Operands 1	2	3
218	pw	-	-

Opcode	Reference	Format	Class
111	varies	varies	110110

Operation: Reads the current processor status and system clock and stores them in a 32-bit location.

Operand 1: A packed word destination operand comprised of the following:

0 - 15: Used to store the processor status.

16 - 31: Used to store the current value of the system clock.

Action:

- Read the 16-bit Processor Status field in the current processor data part, and append it to the current value of the processor resident system clock to form an ordinal value.
- Store the ordinal result in destination operand 1.

INTERCONNECT OPERATORS

MOVE TO INTERCONNECT MOV_TO_ICT

ID#	Operands 1	2	3
219	as	so	so

Opcode	Reference	Format	Class
01111	varies	varies	0010

Operation: Moves a source short-ordinal to a destination interconnect register.

Operand 1: Contains the access selector for the destination interconnect object.

Operand 2: Contains the short-ordinal byte displacement within the interconnect object to the destination interconnect register. This value must be even.

Operand 3: Contains the source short-ordinal that is moved to the destination.

Action: • Move operand 3 to the interconnect register specified by operand 1 and operand 2.

MOVE FROM INTERCONNECT MOV_FM_ICT

ID#	Operands 1	2	3
220	as	so	so

Opcode	Reference	Format	Class
11111	varies	varies	0010

Operation: Moves a source short-ordinal in an interconnect register to a destination short-ordinal.

Operand 1: Contains the access selector for the source interconnect object.

Operand 2: Contains the short-ordinal byte displacement within the interconnect object to the source interconnect register. This value must be even.

Operand 3: Contains the destination short-ordinal that is copied from the source interconnect register.

Action: • Move the contents of the interconnect register specified by operand 1 and operand 2 into short-ordinal operand 3.

BLOCK MOVE OPERATORS

BLOCK MOVE BLK_MOV

ID#	Operands		
	1	2	3
222	pw	pw	so

Opcode	Reference	Format	Class
none	varies	varies	110001

Operation: Moves a block of up to 2,048 bytes of a time, possibly within a larger region being moved using a loop built around this operator.

Operand 1: A packed word operand comprised of the following:
0 - 15: destination access selector
16 - 31: destination displacement

Operand 2: A packed word operand comprised of the following:
0 - 15: source access selector
16 - 31: source displacement

Operand 3: Short ordinal number of bytes to be moved, minus one.

Action:

- The size of the total region to be moved is the number of bytes (operand 3 plus one) rounded up to a multiple of 8.

- The number of bytes that will be moved by this operation is the size of the total region <u>modulo</u> 2,048. (For example, if operand 3 is 4,999, then the size of the total region is 5,000 and the number of bytes that will be moved by this operation is 904.)

- The source region begins with the source displacement in the source object. The destination region begins with the destination displacement in the destination object.

- Move from the top of the source region to the top of the destination region the calculated number of bytes in units of eight bytes. The move operation begins at high displacements and progresses to lower displacements. (For example, if both source and destination displacement are 1,000 and using the sizes from the example above: The bytes in the region to be moved have displacements 1,000 to 5,999. The bytes moved by this operation have displacements 5,096 to 5,099. The first eight-byte unit transferred is from displacements 5,092 to 5,099.) There is no alignment requirement for the source or destination displacements.

This chapter defines GDP instruction composition. It includes the field formats required for the various operand addressing modes and the complete instruction encoding information. For a full discussion of the GDP instruction interface see the Instruction Interface chapter of this manual.

CHAPTER CONVENTIONS

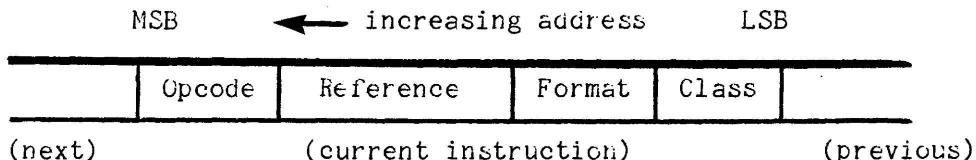
Throughout the tables in this chapter the following abbreviations are used to indicate the lengths of operands:

b	byte (8 bits)
db	double-byte (16 bits)
w	word (32 bits)
dw	double-word (64 bits)
ew	extended-word (80 bits)

Unless otherwise noted, field formats and binary encoded field values are shown in an MSB to LSB order, left to right on the page.

INSTRUCTION FIELDS

Instructions are variable-length sections of a bit-addressed stream in an instruction object. Every instruction contains an operator specification and possibly several references. The operator specifies to the processor what operation is to be performed, and operand references select the operands to be used or manipulated. The major fields of an instruction are ordered as follows:



The operator specified in an instruction is encoded in two fields, the Class field and the Opcode field. The Class field specifies the operator class to which the operator belongs, and the Opcode field selects from within that class the processor operation to be performed. The operator's class determines the order of the operator (i.e., the number of required operands) and the length of the associated operands. Later sections of this chapter define the formats and encodings of the Class, Format, Reference, and Opcode instruction fields.

CLASS FIELD ENCODINGS

ORDER	OPERAND LENGTHS	CLASS ENCODING
0	none	000110
	none (branch)	100110
	none (breakpt)	111111
1	b (branch)	1000
	b	010110
	db	0000
	w	110110
	dw	001110
	ew	101110
2	b, b	011110
	b,db	111110
	b, w	000001
	db, b	100001
	db,db	0100
	db, w	010001
	w, b	001001
	w,db	101001
	w, w	1100
	w,ew	011001
	dw, b	111001
	dw,dw	000101
	dw,ew	100101
	ew, b	010101
	ew, w	110101
	ew,dw	001101
	ew,ew	101101
3	b, b, b	011101
	db,db, b	111101
	db,db,db	0010
	db,db, w	000011
	db, w, w	100011
	db, w,dw	010011
	w,db,db	110011
	w, w, b	001011
	w, w,db	110001
	w, w, w	1010
	w, w,ew	101011
	w,ew,ew	011011
	dw, w, w	111011
	dw,dw, b	000111
	dw,dw,ew	100111
	dw,ew,ew	010111
	ew, w,ew	110111
	ew,dw,ew	001111
	ew,ew, b	101111
	ew,ew,ew	011111

FORMAT FIELD ENCODINGS

The Format field of an instruction determines which references (implicit stack references or explicit data references) in the instruction specify which operands. The following table shows the Format field encodings for the mappings from the possible data or stack references to their associated operands.

ORDER	OPERAND 1	OPERAND 2	OPERAND 3	EXPLICIT REFERENCES	FORMAT ENCODING
0	---	---	---	0	none
1	dref1	---	---	1	0
	stk	---	---	0	1
2	dref1	dref2	---	2	00
	dref1	dref1	---	1	10
	dref1	stk	---	1	01
	stk	dref1	---	1	011
	stk	stk	---	0	111
3	dref1	dref2	dref3	3	0000
	dref1	dref2	dref2	2	1000
	dref1	dref2	dref1	2	0100
	dref1	dref2	stk	2	1100
	dref1	stk	dref2	2	0010
	stk	dref1	dref2	2	1110
	dref1	stk	dref1	1	1010
	stk	dref1	dref1	1	0001
	dref1	stk	stk	1	0110
	stk	dref1	stk	1	1001
	stk1	stk2	dref1	1	0111
	stk2	stk1	dref1	1	0101
	stk1	stk2	stk	0	1011
	stk2	stk1	stk	0	1101
	dref2	dref1	dref3	3	0011
	dref2	dref1	stk	2	1111

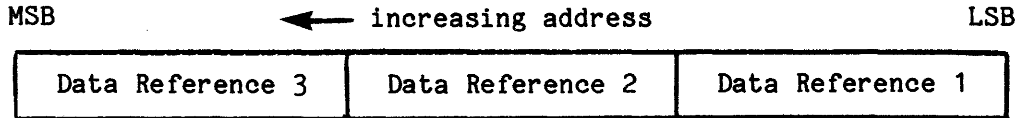
dref1,dref2,dref3
: indicate that the operand is referenced through the first, second, or third explicit data reference in the instruction's reference field.

stk
: indicates that the operand itself is to be pushed onto, or popped from, the operand stack.

stk1,stk2
: indicate that the operand is popped from the top (stk1) or next-to-top (stk2) of the operand stack.

REFERENCE FIELD FORMAT

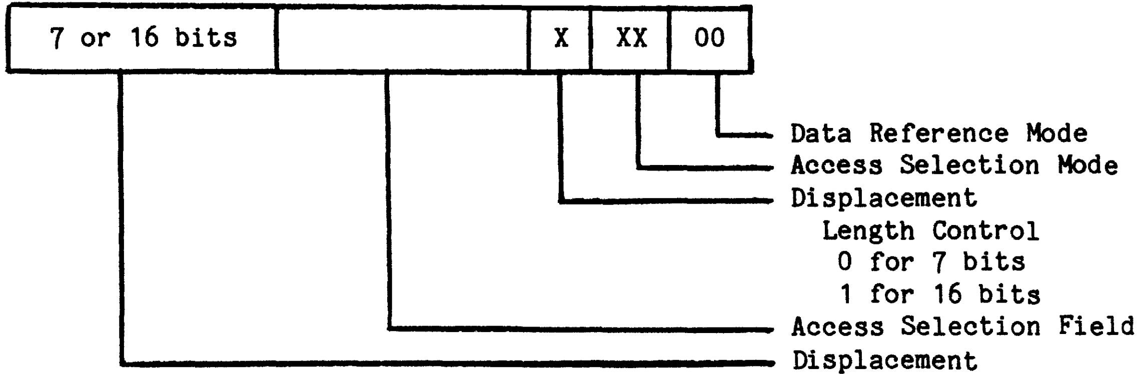
The Reference Field of an instruction can consist of either 0 or 1 data references followed by a branch reference or from 0 to 3 data references. Data references and branch references are both variable in length. Explicit data refererences are encoded in the following order:



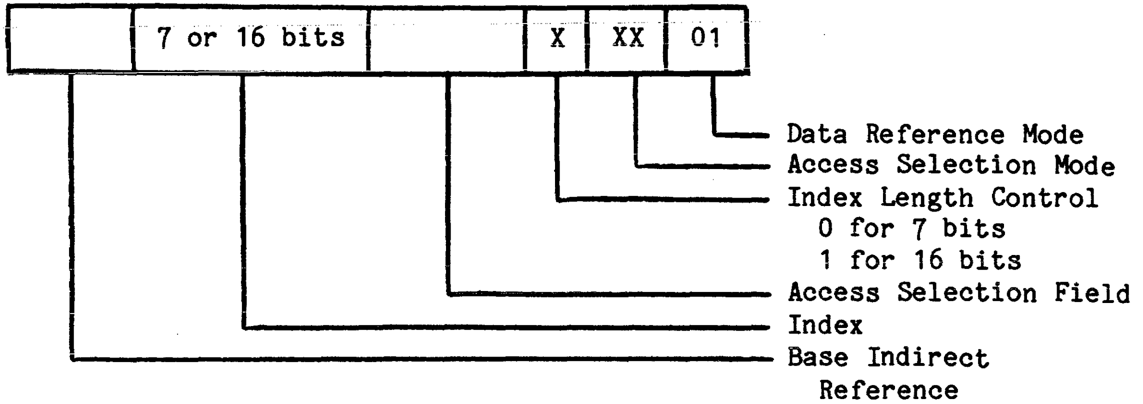
The format of a data reference is determined by the desired data reference mode and access selection mode. The following sections of this chapter define the fields and encodings of a data reference.

DATA REFERENCE FORMATS

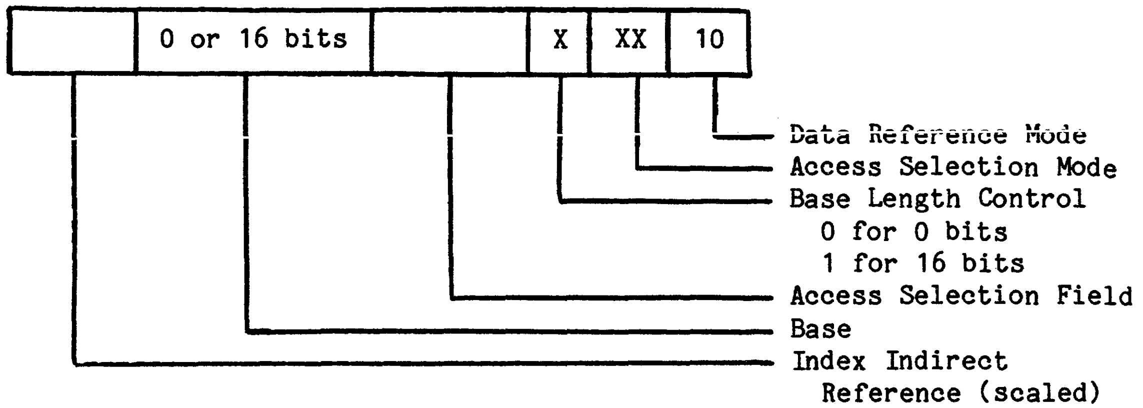
SCALAR DATA REFERENCE



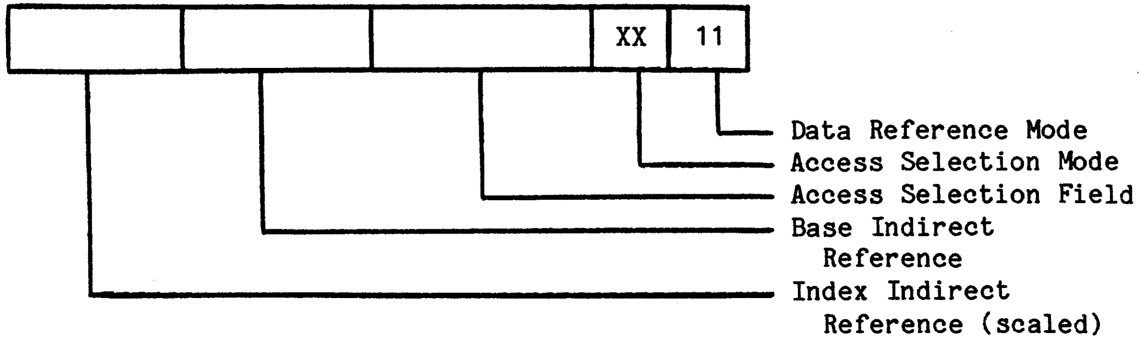
RECORD ITEM DATA REFERENCE



STATIC ARRAY ELEMENT DATA REFERENCE



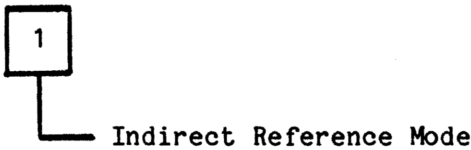
DYNAMIC ARRAY ELEMENT DATA REFERENCE



INDIRECT REFERENCE FIELD FORMATS

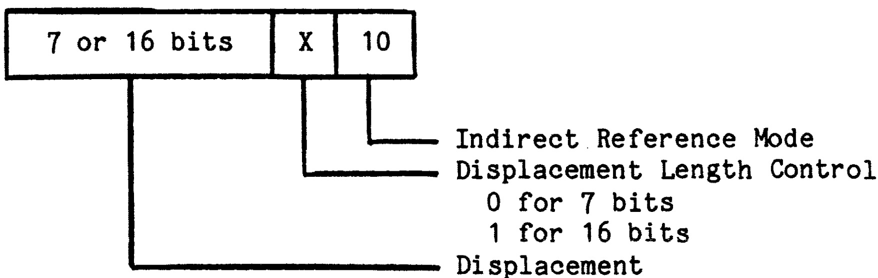
Stack Indirect Reference

The 16-bit indirectly referenced (Base or Index) value is popped from the current top of the operand stack.



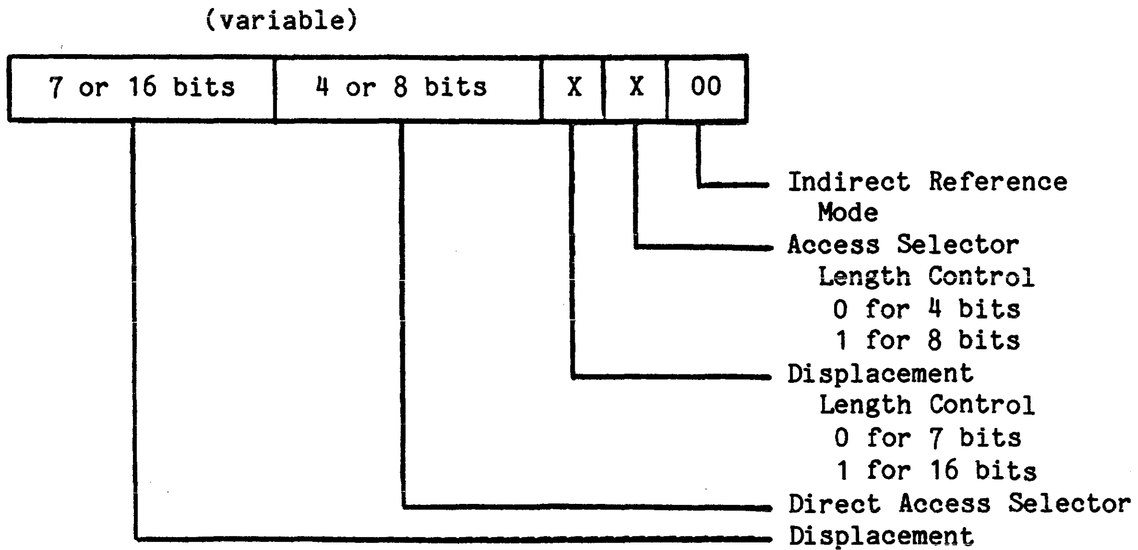
Intrasegment Indirect Reference

The 16-bit indirectly referenced (Base or Index) value is obtained using the below Displacement field to offset into the same data segment that is selected by the Access Selection field of the present data reference.



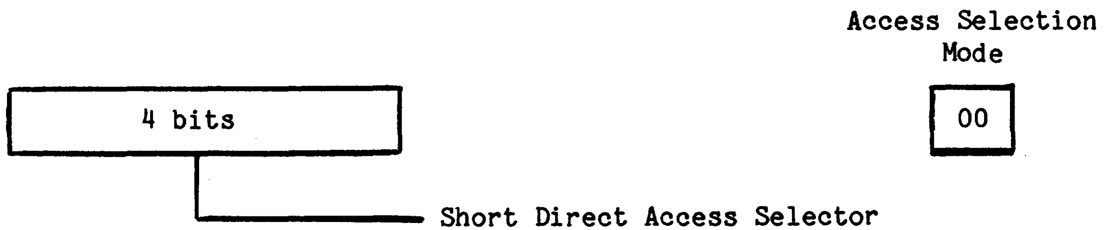
General Indirect Reference

The 16-bit indirectly referenced (Base or Index) value is obtained using the below Displacement field to offset into the data segment selected by the below Access Selector field.

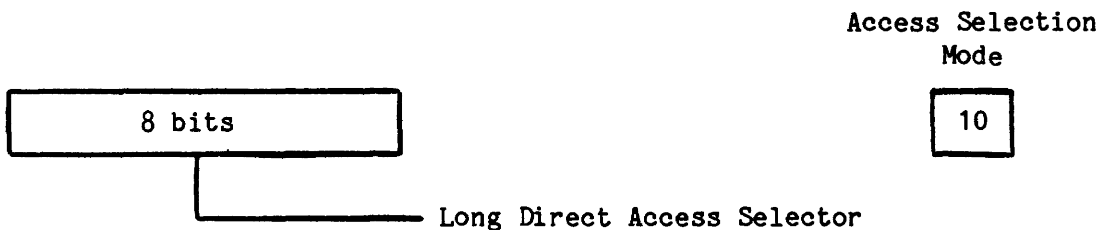


ACCESS SELECTION FIELD FORMATS

Short Direct Access Selection



Long Direct Access Selection



Stack Indirect Access Selection

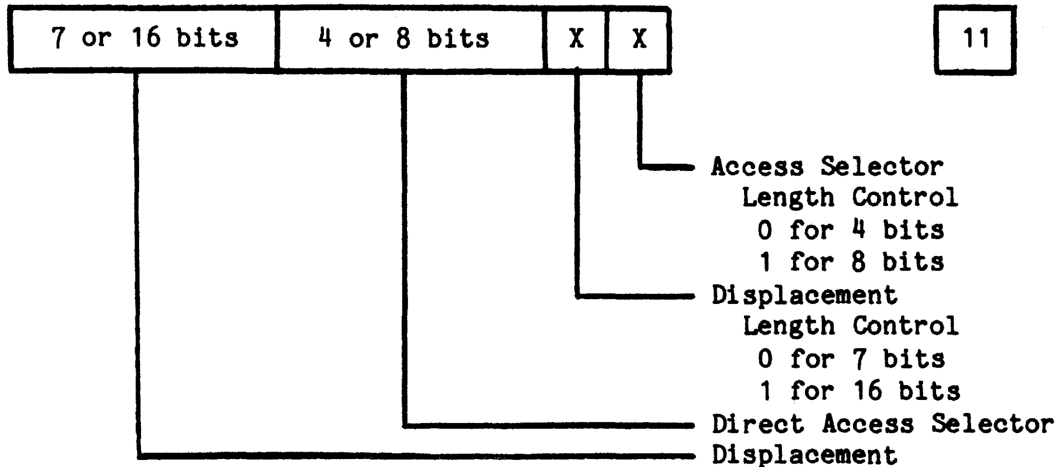
Access Selection Mode: 01

There is no Access Selection field in the instruction stream. Access Selection occurs via a 16-bit indirect access selector that is popped from the current top of the operand stack.

General Indirect Access Selection

Access selection (for the entire data reference) occurs via a 16-bit indirect access selector. The indirect access selector is located, in turn, by using the below Displacement field to offset into the the data object that is selected by the below Direct Access Selector field.

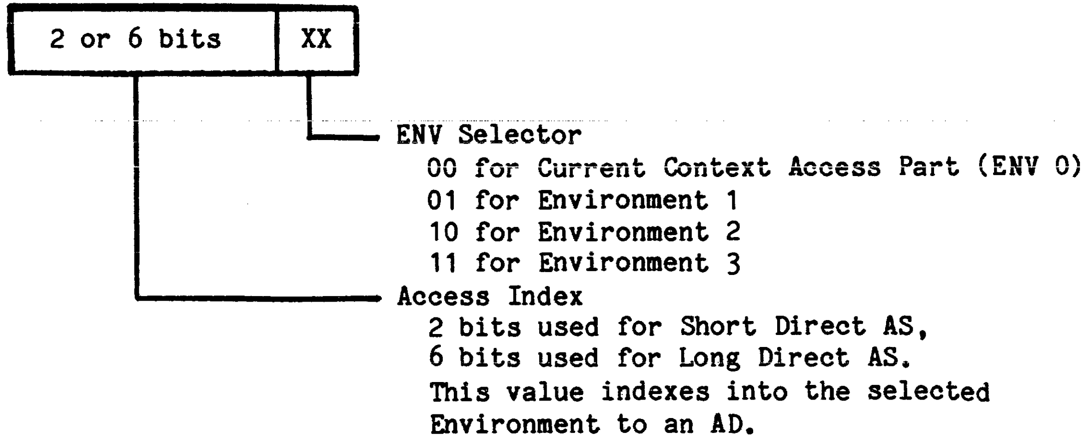
Access Selection Mode: 11



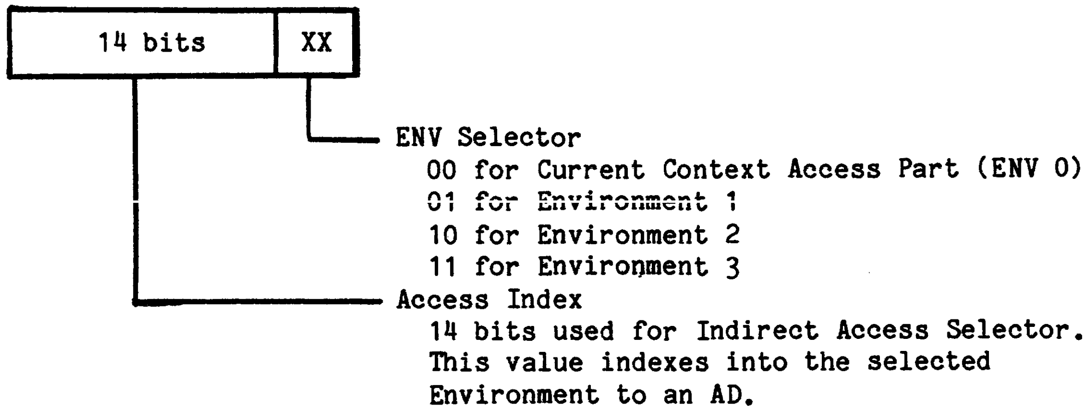
ACCESS SELECTOR FORMATS

An access selector's function is to select an access descriptor. Access selectors in the instruction stream can be either 4 or 8 bits in length. When found in data locations other than the current instruction stream (e.g., when used as operands for object operators), access selectors are always 16 bits in length and are simply called access selectors. Four- or eight-bit access selectors are called direct access selectors when the distinction is required. Access selectors have one of the following formats.

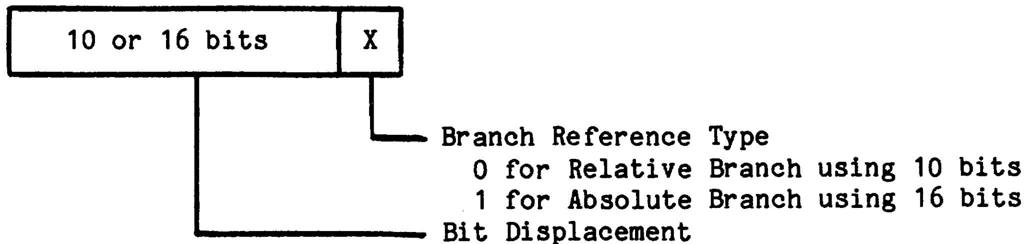
Direct Access Selector



Indirect Access Selector



BRANCH REFERENCE FORMATS



OPCODE ENCODING SUMMARY

OPERATOR	ORDER	OPERAND LENGTHS 1—2—3	CLASS	OPCODE
CHARACTER OPERATORS				
Move Character	2	b, b	011110	00
Zero Character	1	b	010110	0
One Character	1	b	010110	01
Save Character	1	b	010110	11
AND Character	3	b, b, b	011101	000
Inclusive OR Character	3	b, b, b	011101	100
Exclusive OR Character	3	b, b, b	011101	010
Equivalence Character	3	b, b, b	011101	110
NOT Character	2	b, b	011110	10
Add Character	3	b, b, b	011101	001
Subtract Character	3	b, b, b	011101	101
Increment Character	2	b, b	011110	001
Decrement Character	2	b, b	011110	101
Equal Character	3	b, b, b	011101	0011
Not Equal Character	3	b, b, b	011101	1011
Equal Zero Character	2	b, b	011110	011
Not Equal Zero Character	2	b, b	011110	111
Less Than Character	3	b, b, b	011101	0111
Less Than or Equal Character	3	b, b, b	011101	1111
Convert Character to Short Ordinal	2	b,db	111110	none
Convert Character to Integer	2	b, w	000001	none
SHORT-ORDINAL OPERATORS				
Move Short Ordinal	2	db,db	0100	0000
Zero Short Ordinal	1	db	0000	0000
One Short Ordinal	1	db	0000	0100
Save Short Ordinal	1	db	0000	1100
AND Short Ordinal	3	db,db,db	0010	0000
Inclusive OR Short Ordinal	3	db,db,db	0010	1000
Exclusive OR Short Ordinal	3	db,db,db	0010	0100
Equivalence Short Ordinal	3	db,db,db	0010	1100
NOT Short Ordinal	2	db,db	0100	1000
Extract Short Ordinal	3	db,db,db	0010	0010
Insert Short Ordinal	3	db,db,db	0010	1010
Significant Bit Short Ordinal	2	db,db	0100	0100

OPERATOR	ORDER	OPERAND LENGTHS 1—2—3	CLASS	OPCODE
Add Short Ordinal	3	db,db,db	0010	0110
Subtract Short Ordinal	3	db,db,db	0010	1110
Increment Short Ordinal	2	db,db	0100	1100
Decrement Short Ordinal	2	db,db	0100	0010
Multiply Short Ordinal	3	db,db,db	0010	0001
Divide Short Ordinal	3	db,db,db	0010	1001
Remainder Short Ordinal	3	db,db,db	0010	0101
Equal Short Ordinal	3	db,db, b	111101	000
Not Equal Short Ordinal	3	db,db, b	111101	100
Equal Zero Short Ordinal	2	db, b	100001	00
Not Equal Zero Short Ordinal	2	db, b	100001	10
Less Than Short Ordinal	3	db,db, b	111101	010
Less Than or Equal Short Ordinal	3	db,db, b	111101	110
Convert Short Ordinal to Integer	2	db, w	010001	00
SHORT-INTEGER OPERATORS				
Move Short Integer	2	db,db	0100	0000
Zero Short Integer	1	db	0000	000
One Short Integer	1	db	0000	0100
Save Short Integer	1	db	0000	1100
Add Short Integer	3	db,db,db	0010	1101
Subtract Short Integer	3	db,db,db	0010	0011
Increment Short Integer	2	db,db	0100	1010
Decrement Short Integer	2	db,db	0100	0110
Negate Short Integer	2	db,db	0100	1110
Multiply Short Integer	3	db,db,db	0010	01011
Divide Short Integer	3	db,db,db	0010	11011
Remainder Short Integer	3	db,db,db	0010	00111
Equal Short Integer	3	db,db, b	111101	000
Not Equal Short Integer	3	db,db, b	111101	100
Equal Zero Short Integer	2	db, b	100001	00
Not Equal Zero Short Integer	2	db, b	100001	10
Less Than Short Integer	3	db,db, b	111101	001
Less Than or Equal Short Integer	3	db,db, b	111101	101
Positive Short Integer	2	db, b	100001	01
Negative Short Integer	2	db, b	100001	011
Move in Range Short Integer	3	w,db,db	110011	none
Convert Short Integer to Integer	2	db, w	010001	010

OPERATOR	ORDER	OPERAND LENGTHS 1—2—3	CLASS	OPCODE
ORDINAL OPERATORS				
Move Ordinal	2	w, w	1100	000
Zero Ordinal	1	w	110110	00
One Ordinal	1	w	110110	010
Save Ordinal	1	w	110110	110
AND Ordinal	3	w, w, w	1010	000
Inclusive OR Ordinal	3	w, w, w	1010	0100
Exclusive OR Ordinal	3	w, w, w	1010	1100
Equivalence Ordinal	3	w, w, w	1010	0010
NOT Ordinal	2	w, w	1100	100
Extract Ordinal	3	db, w, w	100011	00
Insert Ordinal	3	db, w, w	100011	10
Significant Bit Ordinal	2	w,db	101001	00
Add Ordinal	3	w, w, w	1010	1010
Subtract Ordinal	3	w, w, w	1010	0110
Increment Ordinal	2	w, w	1100	010
Decrement Ordinal	2	w, w	1100	1110
Multiply Ordinal	3	w, w, w	1010	1110
Divide Ordinal	3	w, w, w	1010	0001
Remainder Ordinal	3	w, w, w	1010	1001
Index Ordinal	3	w, w, w	1010	0101
Equal Ordinal	3	w, w, b	001011	000
Not Equal Ordinal	3	w, w, b	001011	100
Equal Zero Ordinal	2	w, b	001001	000
Not Equal Zero Ordinal	2	w, b	001001	100
Less Than Ordinal	3	w, w, b	001011	010
Less Than or Equal Ordinal	3	w, w, b	001011	110
Convert Ordinal to Integer	2	w, w	1100	1110
Convert Ordinal to Temporary Real	2	w,ew	011001	0
INTEGER OPERATORS				
Move Integer	2	w, w	1100	000
Zero Integer	1	w	110110	00
One Integer	1	w	110110	010
Save Integer	1	w	110110	110
Add Integer	3	w, w, w	1010	1101
Subtract Integer	3	w, w, w	1010	0011
Increment Integer	2	w, w	1100	0001
Decrement Integer	2	w, w	1100	1001
Negate Integer	2	w, w	1100	1101
Multiply Integer	3	w, w, w	1010	1011
Divide Integer	3	w, w, w	1010	0111
Remainder Integer	3	w, w, w	1010	1111

OPERATOR	ORDER	OPERAND LENGTHS 1—2—3	CLASS	OPCODE
Equal Integer	3	w, w, b	001011	000
Not Equal Integer	3	w, w, b	001011	100
Equal Zero Integer	2	w, b	001001	000
Not Equal Zero Integer	2	w, b	001001	100
Less Than Integer	3	w, w, b	001011	001
Less Than or Equal Integer	3	w, w, b	001011	101
Positive Integer	2	w, b	001001	010
Negative Integer	2	w, b	001001	110
Move in Range Integer	3	dw, w, w	111011	none
Convert Integer to Character	2	w, b	001001	001
Convert Integer to Short Ordinal	2	w,db	101001	10
Convert Integer to Short Integer	2	w,db	101001	01
Convert Integer to Ordinal	2	w, w	1100	1110
Convert Integer to Temporary Real	2	w,ew	011001	01
SHORT-REAL OPERATORS				
Move Short Real	2	w, w	1100	000
Zero Short Real	1	w	110110	00
Save Short Real	1	w	110110	110
Add Short Real	3	w, w,ew	101011	00
Add Temporary Real to Short Real	3	ew, w,ew	110111	00
Add Short Real to Temporary Real	3	w,ew,ew	011011	00
Subtract Short Real	3	w, w,ew	101011	10
Subtract Temporary Real from Short Real	3	ew, w,ew	110111	10
Subtract Short Real from Temporary Real	3	w,ew,ew	011011	10
Multiply Short Real	3	w, w,ew	101011	01
Multiply Temporary Real by Short Real	3	ew, w,ew	110111	01
Multiply Short Real by Temporary Real	3	w,ew,ew	011011	01
Divide Short Real	3	w, w,ew	101011	11
Divide Temporary Real into Short Real	3	ew, w,ew	110111	11
Divide Short Real into Temporary Real	3	w,ew,ew	011011	11
Negate Short Real	2	w, w	1100	1101
Absolute Value Short Real	2	w, w	1100	0011
Equal Short Real	3	w, w, b	001011	011
Equal Zero Short Real	2	w, b	001001	101
Less Than Short Real	3	w, w, b	001011	0111
Less Than or Equal Short Real	3	w, w, b	001011	1111
Positive Short Real	2	w, b	001001	011
Negative Short Real	2	w, b	001001	111
Convert Short Real to Temporary Real	2	w,ew	011001	11

OPERATOR	ORDER	OPERAND LENGTHS 1—2—3	CLASS	OPCODE
REAL OPERATORS				
Move Real	2	dw,dw	000101	00
Zero Real	1	dw	001110	0
Save Real	1	dw	001110	1
Add Real	3	dw,dw,ew	100111	00
Add Temporary Real to Real	3	ew,dw,ew	001111	00
Add Real to Temporary Real	3	dw,ew,ew	010111	00
Subtract Real	3	dw,dw,ew	100111	10
Subtract Temporary Real from Real	3	ew,dw,ew	001111	10
Subtract Real from Temporary Real	3	dw,ew,ew	010111	10
Multiply Real	3	dw,dw,ew	100111	01
Multiply Temporary Real by Real	3	ew,dw,ew	001111	01
Multiply Real by Temporary Real	3	dw,ew,ew	010111	01
Divide Real	3	dw,dw,ew	100111	11
Divide Temporary Real into Real	3	ew,dw,ew	001111	11
Divide Real into Temporary Real	3	dw,ew,ew	010111	11
Negate Real	2	dw,dw	000101	10
Absolute Value Real	2	dw,dw	000101	01
Equal Real	3	dw,dw, b	000111	0
Equal Zero Real	2	dw, b	111001	0
Less Than Real	3	dw,dw, b	000111	01
Less Than or Equal Real	3	dw,dw, b	000111	11
Positive Real	2	dw, b	111001	01
Negative Real	2	dw, b	111001	11
Convert Real to Temporary Real	2	dw,ew	100101	none
TEMPORARY-REAL OPERATORS				
Move Temporary Real	2	ew,ew	101101	00
Zero Temporary Real	1	ew	101110	0
Save Temporary Real	1	ew	101110	1
Add Temporary Real	3	ew,ew,ew	011111	00
Subtract Temporary Real	3	ew,ew,ew	011111	10
Multiply Temporary Real	3	ew,ew,ew	011111	01
Divide Temporary Real	3	ew,ew,ew	011111	011
Remainder Temporary Real	3	ew,ew,ew	011111	111
Negate Temporary Real	2	ew,ew	101101	10
Square Root Temporary Real	2	ew,ew	101101	01
Absolute Value Temporary Real	2	ew,ew	101101	11

OPERATOR	ORDER	OPERAND LENGTHS 1—2—3	CLASS	OPCODE
Equal Temporary Real	3	ew,ew, b	101111	0
Equal Zero Temporary Real	2	ew, b	010101	0
Less Than Temporary Real	3	ew,ew, b	101111	01
Less Than or Equal Temporary Real	3	ew,ew, b	101111	11
Positive Temporary Real	2	ew, b	010101	01
Negative Temporary Real	2	ew, b	010101	11
Convert Temporary Real to Ordinal	2	ew, w	110101	0
Convert Temporary Real to Integer	2	ew, w	110101	01
Convert Temporary Real to Short Real	2	ew, w	110101	11
Convert Temporary Real to Real	2	ew,dw	001101	none
OBJECT OPERATORS				
Branch	0	none	100110	none
Branch True	1	b	1000	0
Branch False	1	b	1000	1
Branch Indirect	1	db	0000	0010
Branch Intersegment	1	w	110110	001
Branch Intersegment without Trace	1	w	110110	101
Branch Intersegment and Link	2	w, w	1100	1011
Breakpoint	0	none	111111	none
Copy Access Descriptor	2	db,db	0100	0001
Null Access Descriptor	1	db	0000	1010
Amplify Rights	2	db,db	0100	1001
Restrict Rights	2	w,db	101001	011
Retrieve Type Definition	2	db,db	0100	0101
Create Refinement	3	db, w,dw	010011	none
Create Typed Refinement	2	dw,dw	000101	11
Create Object	3	db,db, w	000011	0
Create Typed Object	3	db, w, w	100011	01
Inspect Access Descriptor	2	db, w	010001	110
Inspect Object	2	db, w	010001	001
Equal Access	3	db,db, b	111101	0011
Move to Embedded Data Value	2	w,db	101001	111
Move from Embedded Data Value	2	db, w	010001	110
Lock Object	3	db,db, b	111101	1011
Unlock Object	2	db,db	0100	1101
Indivisibly Add Short Ordinal	2	db,db	0100	0011
Indivisibly Add Ordinal	2	w, w	1100	0111
Indivisibly Insert Short Ordinal	3	db,db,db	0010	10111
Indivisibly Insert Ordinal	3	db, w, w	100011	011

OPERATOR	ORDER	OPERAND LENGTHS 1—2—3	CLASS	OPCODE
Enter Environment 1	1	db	0000	0110
Enter Environment 2	1	db	0000	1110
Enter Environment 3	1	db	0000	0001
Copy Process Globals	1	db	0000	1001
Set Context Mode	1	db	0000	0101
Adjust Stack Pointer	2	db,db	0100	01011
Call	2	db, w	010001	101
Call Through Domain	2	db, w	010001	011
Return	0	none	000110	0
Return and Fault	0	none	000110	1
Send	2	db,db	0100	01111
Receive	1	db	0000	1011
Conditional Send	3	db,db, b	111101	0111
Conditional Receive	2	db, b	100001	111
Surrogate Send	3	db,db, w	000011	1
Surrogate Receive	2	db, w	010001	111
Delay Process	1	db	0000	0111
Send Process	1	db	0000	1111
Set Process Mode	2	db,db	0100	11111
Read Process Clock	1	w	110110	011
Send to Processor	3	db,db, b	111101	1111
Read Processor Status	1	w	110110	111
Move to Interconnect	3	db,db,db	0010	01111
Move from Interconnect	3	db,db,db	0010	11111
Block Move	3	w, w,db	110001	none

Where:

- b byte (8 bits)
- db double-byte (16 bits)
- w word (32 bits)
- dw double-word (64 bits)
- ew extended-word (80 bits)

This chapter provides reference information for the GDP's support of Faulting and Tracing. The Fault and Trace Areas are defined and fault encoding are listed.

FAULT REFERENCE

Faults can occur at the following severity levels (the least severe first):

- Context-level fault: These faults require interruption of the normal execution of instructions within the currently active context. When such a fault occurs, information identifying its cause is recorded by the processor in the Context Fault Data Area of the process object. The Context Faulted bit in the process status is set to 1. An intersegment branch is then effectively executed to offset 64 of the Fault Instruction Object currently designated as context fault handler (referenced by AD 0 in the defining domain).

- Process-level fault: These faults require suspension of the faulted process and repair by a fault handling process. When such a fault occurs, the processor is preempted from executing the currently running process and information about the fault is recorded in the Process Fault Data area in the process carrier of the faulted process. The Process Faulted bit in the process status is set to 1. The Second Port AD in the process carrier is copied into the Save Port AD location of the process carrier. The carrier of the preempted process is then sent as a message to the Fault Port referenced by its process object. The processor then attempts normal dispatching at its current dispatching port.

- Processor-level fault: The most severe fault disruption of processing is a processor-level fault. Such a fault requires the suspension of both the executing process and the normal dispatching mechanism for the processor. When such a fault occurs, information about the fault is recorded in the Processor Fault Access and Processor Fault Data areas of the faulted processor object. If there is a process associated with the processor which is not process-faulted, the execution of the process is preempted and it is forwarded to the second port in its carrier. If the current process is process-faulted, the Message Received bit in the processor Carrier Status is set to 1 and the incoming message of the processor carrier is set to reference the current process. The processor dispatching mode is then switched from Normal to Diagnostic and the processor attempts dispatching at the Diagnostic Dispatching Port (referenced in the processor object).

FAULT AREA FORMATS

Fault areas are used to record appropriate fault information. The information can be used by fault handling software to determine the nature of the fault and to administer recovery. Each Fault Area has a fixed format which defines fields for any information that may be stored by faults at a specific level. Which fields actually contain valid information after a fault depends on the specific fault.

PROCESSOR FAULT ACCESS AREA

The Processor Fault Access Area in the processor object is organized as follows:

	Access Descriptor Index (32 bits each)
AD to Current Carrier	8
AD to Current Port	9

The fields have the following meanings:

Current Carrier (AD 8)
This AD references the carrier being operated on by a processor port operation when the fault occurred.

Current Port (AD 9)
This AD references the port being operated on by a processor port operation when the fault occurred.

PROCESS FAULT ACCESS AREA

The Process Fault Access Area in the process access part is organized as follows:

	Access Descriptor Index (32 bits each)
AD to Current Carrier or New AD	8
AD to Current Port	9
AD to Current Surrogate Carrier	10

The fields have the following meanings:

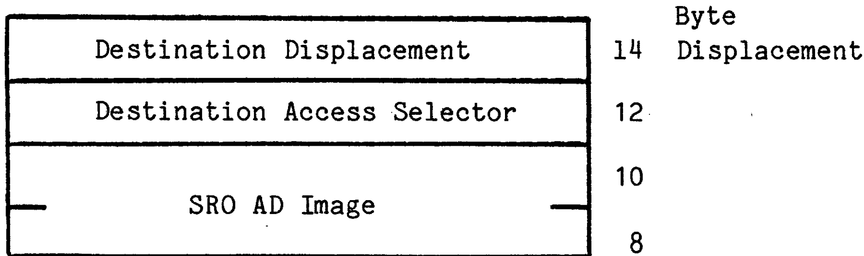
Current Carrier or New AD (AD 8)
As Current Carrier, this AD references the carrier being operated on by a port operation when the fault occurred. As New AD, this field is written by the processor when an allocation-related fault occurs. It then contains the associated access descriptor of the newly allocated OD.

Current Port (AD 9)
This AD references the port being operated on by a port operation when the fault occurred.

Current Surrogate Carrier (AD 11)
This AD references the unrefined surrogate carrier specified by an interprocess SURROGATE SEND or SURROGATE RECEIVE instruction that faulted.

ALLOCATION FAULT AREA

The Allocation Fault Area in the Process Object is organized as follows:



The fields that constitute the Allocation Fault Area have the following meanings:

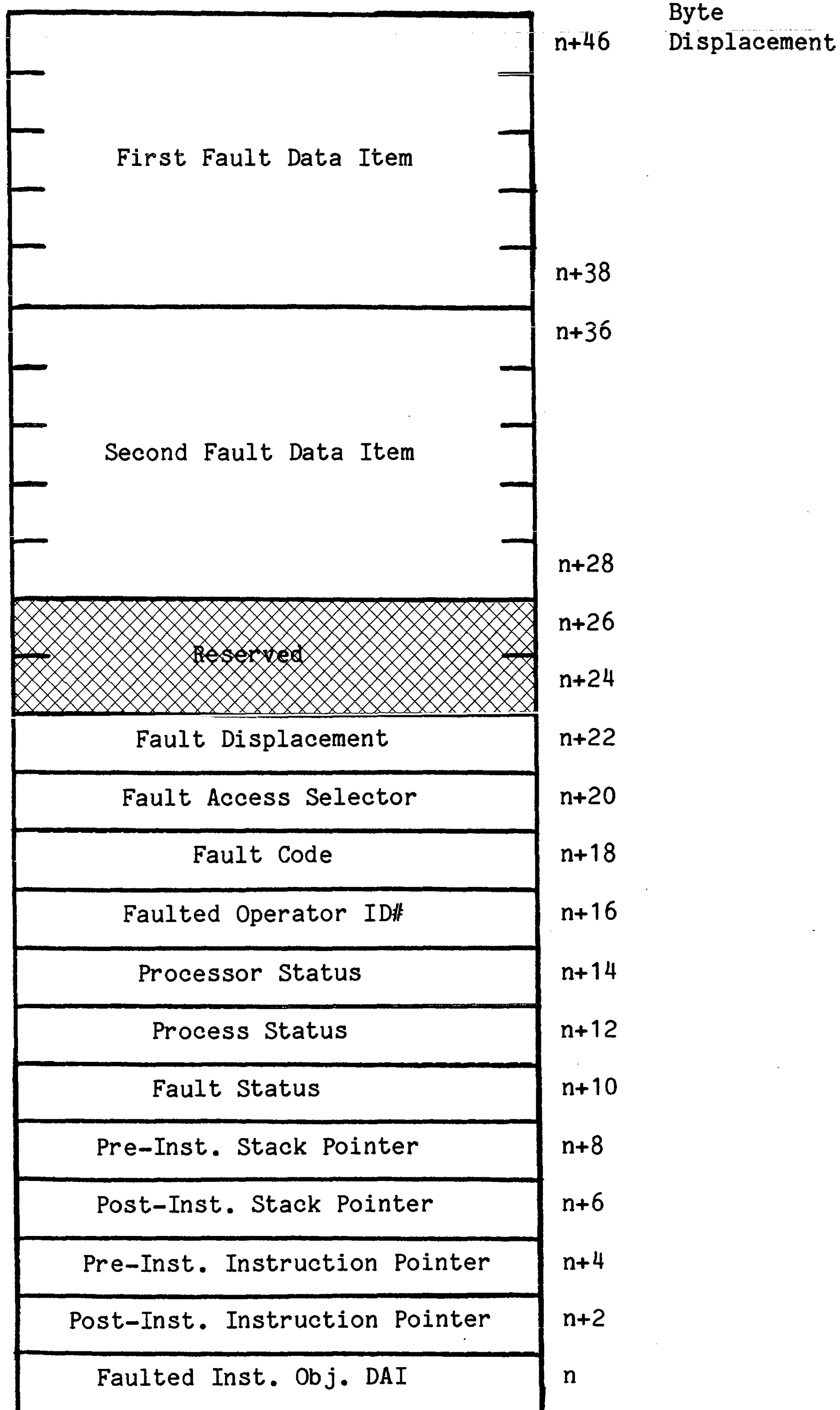
SRO AD Image (Bytes 8 - 11)
: This 32-bit field is written by the processor when an allocation-related fault occurs. It then contains the image of the associated access descriptor for the specified SRO. A value of zero indicates the process allocation stack was specified.

Destination Access Selector (Bytes 12 - 13)
: This 16-bit field is written by processor when an allocation-related fault occurs. It is the access selector for the destination AD associated with the newly allocated object descriptor. In certain operators, this field contains the access selector of the data object where a boolean result should be stored.

Destination Displacement (Bytes 14 - 15)
: In some operators, this 16-bit field contains the displacement of the boolean result.

FAULT DATA AREA

The Fault Data Area is a 48-byte record organized as follows:



The Fault Data Area for context-, process-, and processor-level faults has the same organization (shown above). Process objects contain Fault Data Areas for context- and process-level faults. Processor objects contain Fault Data Areas for processor-level faults. The fields in the Fault Data Area are interpreted as follows:

Faulted Inst. Obj. DAI (Bytes n thru n+1)
Records the DAI for the instruction object in which the faulted instruction is located.

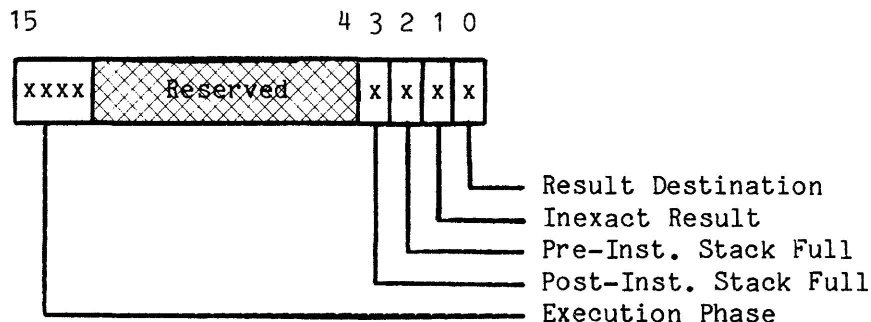
Post-Inst. Instruction Pointer (Bytes n+2 thru n+3)
Records the instruction pointer of the instruction physically following the instruction that caused the fault. If the fault occurred during instruction decoding, this field is undefined.

Pre-Inst. Instruction Pointer (Bytes n+4 thru n+5)
Records the instruction pointer of the instruction which caused the fault.

Post-Inst. Stack Pointer (Bytes n+6 thru n+7)
Records the operand stack pointer at the time the fault occurred. This value should be incremented by 2 if the Post-Inst. Stack Full bit in the Fault Status is 1.

Pre-Inst. Stack Pointer (Bytes n+8 thru n+9)
Records the operand stack pointer at the beginning of the instruction that caused the fault. This value should be incremented by 2 if the Pre-Inst. Stack Full bit in the Fault Status is 1.

Fault Status (Bytes n+10 thru n+11)
The Fault Status field has the following organization:



These fields are interpreted as follows:

Result Destination (Bit 0)
This bit records where the operand destination should have been:
0 - Destination was the operand stack
1 - Destination was in memory

Inexact Result (Bit 1)

This bit records whether the generated result was exact or inexact:

0 - exact
1 - inexact

Pre-Inst. Stack Full (Bit 2)

This bit records whether the 16-bit top of stack register (within the GDP) was occupied at the beginning of the faulted instruction:

0 - empty
1 - occupied

Post-Inst. Stack Full (Bit 3)

This bit records whether the top-of-stack register (within the GDP) was occupied when the instruction faulted:

0 - empty
1 - occupied

Execution Phase (Bits 12 - 15)

This 4-bit field records a value that indicates the phase of execution when the fault occurred. It is used to identify fault handling strategies in the more complex operators. A value of zero indicates that the instruction can be re-executed with no fault handling repair of data necessary.

Process Status (Bytes n+12 thru n+13)

This 16-bit field records the process status at the time the fault occurred.

Processor Status (Bytes n+14 thru n+15)

This 16-bit field records the processor status at the time the fault occurred.

Faulted Operator ID# (Bytes n+16 thru n+17)

If the fault occurred during instruction decoding, this field is zero. Otherwise, this field records the operator ID# of the faulted instruction.

Fault Code (Bytes n+18 thru n+19)

The Fault Code field contains a processor-written 16-bit encoding that indicates the specific fault that occurred. The detailed encodings of this field are defined in subsequent sections of this chapter.

Fault Access Selector (Bytes n+20 thru n+21)

The interpretation of this field varies depending on the specific fault. See the following sections of this chapter for more details.

Fault Displacement (Bytes n+22 thru n+23)

The interpretation of this field varies depending on the specific fault. See the following sections of this chapter for more details.

Second Fault Data Item (Bytes n+28 thru n+37)
The value in this field depends on whether the fault is pre-operation or post-operation:

- If the fault is pre-operation, this field contains the value of the source operand 1. Unused high-order bits are undefined.
- If the fault is post-operation, this field is not defined.

First Fault Data Item (Bytes n+38 thru n+47)
The value in this field depends on whether the fault is pre-operation or post-operation:

- If the fault is pre-operation, this field contains the value of the source operand 2. Unused high-order bits are undefined.
- If the fault is post-operation, this field contains the value of the exceptional result. Unused high-order bits are undefined.

PROCESS FAULT RESTART AREA

The Process Fault Restart Area in a Process Carrier Object is organized as follows:

Field	Byte Displacement
Instruction Pointer	22
Instruction Object DAI	20
Operand Stack Pointer	18
Restart Status	16

The fields that constitute the Process Fault Restart Area have the following meanings:

Restart Status (Bytes 16 - 17)
The lower bit of this field is the Restart Boolean and determines whether the next 3 fields (Operand Stack Pointer, Instruction Object DAI, and Instruction Pointer) should be copied by the GDP into the corresponding entries of the current context when the process is restarted. The Restart Boolean also determines whether the entire Restart Status field will be copied into the Process Status for the process. (The Restart Boolean bit itself will thus be copied into the Unbound bit in the Process Status; the Unbound bit is cleared when the process is bound to the processor). If the Restart Boolean is false, no copy is performed. Otherwise, the three context fields and one process field are copied, then the Restart Boolean bit is cleared by the processor. This allows restarting a faulted process without changing the current context.

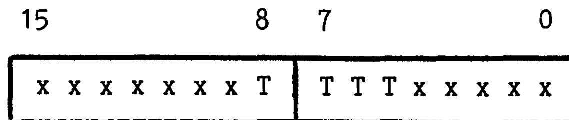
Operand Stack Pointer (Bytes 18 - 19)
Instruction Object DAI (Bytes 20 - 21)
Instruction Pointer (Bytes 22 - 23)

These three fields have the same interpretation as the corresponding fields in the context object.

FAULT CODES

FAULT TYPES

Faults are categorized into seven general types as determined by bits 5 through 8 of the Fault Code field:



In subsequent encoding diagrams in this chapter, the x values designate bits that are undefined for the particular fault type being described. The TTTT bits shown above are used to encode the general type of the fault that occurred. In addition to the TTTT bits, certain of the other bits in the Fault Code are used to further encode the nature of the fault.

The following list defines the TTTT encodings and gives a two letter mnemonic for the fault type. These mnemonics are used throughout this chapter.

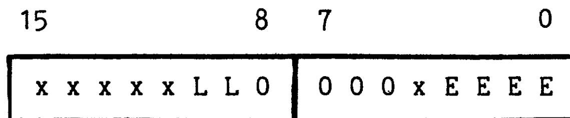
TYPE	TTTT	MNEM	Faults
0	0000	FF	All other faults not named here
1	0001	IP	Instruction Pointer Overflow fault
2	0010	TS	Test Object Type or Entry Type faults
4	0100	SO	Segment Overflow fault
5	0101	MO	Memory Overflow fault (physical addr >= 2**24)
6	0110	RR	Read Rights fault
7	0111	WR	Write Rights fault

TTTT values 3 and 8 through 15 are undefined.

All faults of types 1 through 7 are process-level faults. Subsequent sections of this chapter describe the more detailed fault encodings for the different fault types.

Type 0 Faults

Type 0 faults have the following bits defined in the Fault Code field:



The LL bits encode the fault level as follows:

LL	Description
00	Context-Level Faults
01	Process-Level Faults (group 1)
10	Process-Level Faults (group 2)
11	Processor-Level Faults

The EEEE bits encode the specific fault within the level group.

The following Type 0 Fault List presents the type 0 faults in the order of their encoding. The encoding column of this table (and of other tables in the following sections) contains the LL EEEE bits if the type is 0 (FF).

TYPE 0 FAULT LIST

FAULTS	TYPE	ENCODING LL EEEE
Domain Error Fault	0 (FF)	00 0000
Overflow Fault	0 (FF)	00 0001
Underflow Fault	0 (FF)	00 0010
Inexact Fault	0 (FF)	00 0011
Return Fault	0 (FF)	00 0100
		LL EEEE
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Fault	0 (FF)	01 0001
Domain Access Index Overflow Fault	0 (FF)	01 0010
Destination Delete Rights Fault	0 (FF)	01 0011
Race Condition Fault	0 (FF)	01 0011
Level Fault	0 (FF)	01 0100
Access Path Object Descriptor Validity Faults	0 (FF)	01 0101
Instruction Object Type Rights Fault	0 (FF)	01 0101
Odd Interconnect Descriptor Base Address Fault	0 (FF)	01 0101
Source AD Validity Fault	0 (FF)	01 0101
Surrogate Carrier Validity	0 (FF)	01 0101
Surrogate Carrier Type Rights Fault	0 (FF)	01 0101
Context Parameters Size Faults	0 (FF)	01 0110
TCO Type Rights Fault	0 (FF)	01 0110
Odd Displacement Fault	0 (FF)	01 0110
Port Type Rights Fault	0 (FF)	01 0110
PCO Type Rights Fault	0 (FF)	01 0110
Process Object Type Rights Fault	0 (FF)	01 0110
Source Representation Rights Fault	0 (FF)	01 0110
Context Object Type Rights Fault	0 (FF)	01 0111
SRO Type Rights Fault	0 (FF)	01 0111
Destination Port Type Rights Fault	0 (FF)	01 0111
Clear Memory Size Fault	0 (FF)	01 1000
Process Object Access Mismatch Fault	0 (FF)	01 1000
Type Fault	0 (FF)	01 1000
Carrier Lock Fault	0 (FF)	01 1001
Object Lock ID/Type Fault	0 (FF)	01 1001
Offset and Length Compatibility Fault	0 (FF)	01 1001
PSO Lock Fault	0 (FF)	01 1001
SRO Lock Fault	0 (FF)	01 1001
Domain Access Index Overflow Fault	0 (FF)	01 1010
Port Lock Fault	0 (FF)	01 1010
Refinement Overflow Fault	0 (FF)	01 1010
Object Descriptor Exhaustion Fault	0 (FF)	01 1011
Carrier Queued Fault	0 (FF)	01 1100
Storage Block Index Overflow Fault	0 (FF)	01 1101
Storage Block Fragmentation Fault	0 (FF)	01 1110
Storage Claim Underflow Fault	0 (FF)	01 1111
		LL EEEE
Instruction Fetch Fault	0 (FF)	10 0000
Instruction Object Displacement Fault	0 (FF)	10 0000

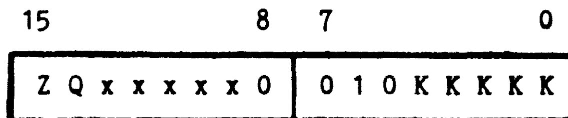
FAULTS	TYPE	ENCODING LL EEEE
Bus Error Fault	0 (FF)	11 0000
Process Level Objects Lock Fault	0 (FF)	11 0001
Process Lock Fault	0 (FF)	11 0001
PCO Response Count Fault	0 (FF)	11 0010
PCO Lock Fault	0 (FF)	11 0011

Type 1 Faults

Type 1 faults have only the TTTT bits defined in the Fault Code field to distinguish it. A Type 1 fault is for an instruction pointer overflow during a relative branch.

Type 2 Faults

Type 2 faults have the following bits defined in the Fault Code field:



The Z bit indicates whether the fault resulted with testing the object type or the object table entry type. The Z bit is defined as follows:

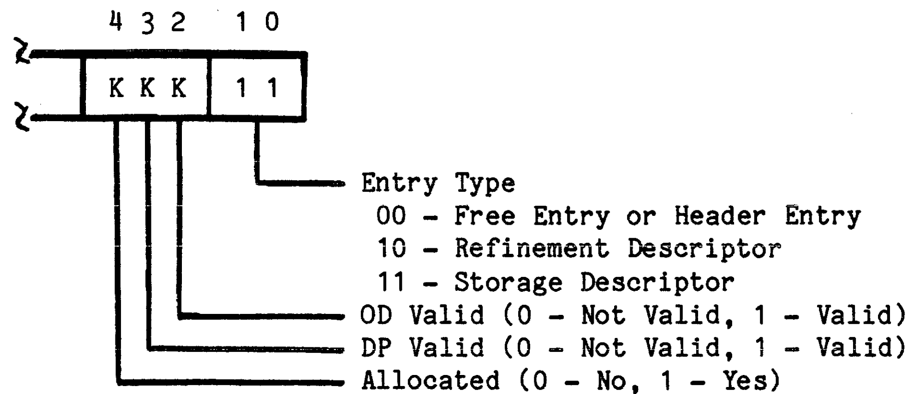
0 - OTE type test
1 - Object type test

The Q bit indicates whether the fault is associated with object table qualification. It thus determines the meaning of the Fault Access Selector and Fault Displacement fields in the fault data area as follows:

0 - The fault did not occur during object table qualification and the Fault Access Selector and Fault Displacement fields contain the indices in the associated access descriptor.
1 - The fault occurred during object table qualification and the Fault Displacement field contains the directory index.

The Z bit determines two alternate interpretations of the KKKKK bits as follows:

Z=0 (fault because of object table entry type test). The KKKKK bits encode the expected values of the least-significant 5 bits of the object table entry. Their meanings are thus determined by the expected Entry Type of the object table entry. The following case is for a storage descriptor:



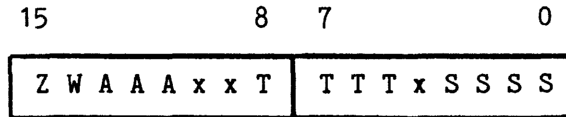
Z=1 (fault because of system type test). The KKKKK bits encode the expected value of the System Type field in the faulted object table entry:

KKKKK	SYSTEM TYPE
00000	Generic object
00001	Object Table Object
00010	Domain Object
00011	Instruction Object
00100	Context Object
00101	Process Object
00110	Processor Object
00111	Port Object
01000	Carrier Object
01001	Storage Resource Object
01010	Physical Storage Object
01011	Storage Claim Object
01100	Dynamic Type Object
01101	Type Definition Object
01110	Type Control Object
01111	RESERVED
10000	Processor Communication Object
10001 thru 11111	RESERVED

The encoding column of the tables in the later sections of this chapter contains the Z KKKKK bits if the type is 2 (TS).

Types 4,5,6,7 Faults

These faults have the following bits defined in the Fault Code field:



These fault types are memory access faults. The W bit indicates whether the fault occurred on a read or write:

0 - Faulted on Read
1 - Faulted on Write

The AAA and Z bits indicate the type of memory access that faulted:

AAA	Z	TYPE OF ACCESS
0xx	x	Storage Address Space (Data Part) The storage segment being accessed is indicated by the SSSS bits. Displacement is given by the Fault Displacement field in the Fault Data Area.
110	0	Storage Address Space (Access Part) The storage segment being accessed is indicated by the SSSS bits. Displacement is given by the Fault Displacement field in the Fault Data Area.
100	x	Access Environment The access selector of the segment is given by the Fault Access Selector in the Fault Data Area.
110	1	Interconnect Address Space Displacement is given by the Fault Displacement field in the Fault Data Area.
111	x	Operand Stack Displacement is given by the Post-Inst. Stack Pointer field in the Fault Data Area.

The SSSS bits are defined for accesses to both the storage address space (data part or access part) and interconnect address space. The SSSS bits are defined as follows:

SSSS	OBJECT BEING ACCESSED
0000	Context Access Part
0101	Processor Object
0111	Process Object
1000	Instruction Object
1010	Defining Domain
1011	Process Carrier
1100	Context Data Part
1101	Object Table Directory
1110	Object Cache (The Fault Access Selector field contains the Access Selector of the object).
1111	Object Table Cache (The Fault Access Selector field bits 4-15 contains the directory index from the AD).

SSSS values 0001, 0010, 0011, 0100, 0110, and 1001 are undefined.

GENERAL FAULT GROUPS

The following faults can occur anywhere during the execution of an operator or sub-operation (which includes instruction decoding, process dispatching, binding etc.). These faults are not explicitly referenced in the later sections. The => symbol indicates that the group name preceding it stands for any of the possible faults that are listed after it. A group name is used in this table (and others in this chapter) by enclosing the name in angle brackets <like so>. This indicates that any of the possible faults of that named group are included.

FAULT GROUPS	TYPE	ENCODING
Memory Reference Faults =>		
Segment Overflow Fault	4 (SO)	
Memory Overflow Fault	5 (MO)	
Read Rights Fault	6 (RR)	
Write Rights Fault	7 (WR)	
Bus Error	0 (FF)	01 0010
Instruction Fetch Fault	0 (FF)	10 0000
Data Part Cache Qualification Faults =>		
Data Part Altered Faults =>		
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Type Fault	2 (TS)	0 11111
	2 (TS)	0 01110
Object Table Cache Qualification Faults =>		
Object Descriptor Type Fault	2 (TS)	0 11111
Object Type Fault	2 (TS)	1 00001
Access Environment Altered Faults =>		
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Fault	0 (FF)	01 0001

DATA OPERATOR FAULT GROUPS

FAULT GROUP	TYPE	ENCODING LL EEEE-
Domain Error Fault	0 (FF)	00 0000
Overflow Fault	0 (FF)	00 0001
Underflow Fault	0 (FF)	00 0010
Inexact Fault	0 (FF)	00 0011

DATA OPERATOR FAULTS

The following table defines the data operator faults for the GDP. The R and P columns indicate whether or not the operator makes use of the Rounding Control bits or the Precision Control bits, respectively, in the context status. An x in a column indicates that the operator does make use of the corresponding control bits. A number in parentheses following a fault type in data manipulation operators specifies the nature of the exceptional result that is produced when that post-operation fault occurs. These numbered notes are defined as follows:

(1) Exceptional result (in the First Fault Data Item) is undefined.
(2) Exceptional result (in the First Fault Data Item) is the single source operand.
(3) Exceptional result (in the First Fault Data Item) has correct significand with wrapped-around exponent.
(4) The fault occurs only if the inexact control bit in the process status is set. The First Fault Data Item contains the exceptional result.

OPERATOR	R	P	FAULTS	COMMENT
Move Character	-	-	None	
Zero Character	-	-	None	
One Character	-	-	None	
Save Character	-	-	None	
AND Character	-	-	None	
Inclusive OR Character	-	-	None	
Exclusive OR Character	-	-	None	
Equivalence Character	-	-	None	
NOT Character	-	-	None	
Add Character	-	-	Overflow(1)	True result is > 255
Subtract Character	-	-	Overflow(1)	True result is < 0
Increment Character	-	-	Overflow(1)	Attempt to increment 255
Decrement Character	-	-	Overflow(1)	Attempt to decrement 0
Equal Character	-	-	None	
Not Equal Character	-	-	None	
Equal Zero Character	-	-	None	
Not Equal Zero Character	-	-	None	
Less Than Character	-	-	None	
Less Than or Equal Character	-	-	None	
Convert Character to Short Ordinal	-	-	None	
Convert Character to Integer	-	-	None	

OPERATOR	R	P	FAULTS	COMMENT
Move Short Ordinal	-	-	None	
Zero Short Ordinal	-	-	None	
One Short Ordinal	-	-	None	
Save Short Ordinal	-	-	None	
AND Short Ordinal	-	-	None	
Inclusive OR Short Ordinal	-	-	None	
Exclusive OR Short Ordinal	-	-	None	
Equivalence Short Ordinal	-	-	None	
NOT Short Ordinal	-	-	None	
Extract Short Ordinal	-	-	None	
Insert Short Ordinal	-	-	None	
Significant Bit Short Ordinal	-	-	None	
Add Short Ordinal	-	-	Overflow(1)	True result is > 65,535
Subtract Short Ordinal	-	-	Overflow(1)	True result is < 0
Increment Short Ordinal	-	-	Overflow(1)	Attempt to increment 65,535
Decrement Short Ordinal	-	-	Overflow(1)	Attempt to decrement 0
Multiply Short Ordinal	-	-	Overflow(1)	True result is > 65,535
Divide Short Ordinal	-	-	Domain Error	Division by zero
Remainder Short Ordinal	-	-	Domain Error	Division by zero
Equal Short Ordinal	-	-	None	
Not Equal Short Ordinal	-	-	None	
Equal Zero Short Ordinal	-	-	None	
Not Equal Zero Short Ordinal	-	-	None	
Less Than Short Ordinal	-	-	None	
Less Than or Equal Short Ordinal	-	-	None	
Convert Short Ordinal to Integer	-	-	None	

OPERATOR	R	P	FAULTS	COMMENT
Move Short Integer	–	–	None	
Zero Short Integer	–	–	None	
One Short Integer	–	–	None	
Save Short Integer	–	–	None	
Add Short Integer	–	–	Overflow(1)	True result is > 32,767 or < -32,768
Subtract Short Integer	–	–	Overflow(1)	True result is > 32,767 or < -32,768
Increment Short Integer	–	–	Overflow(1)	Attempt to increment 32,767
Decrement Short Integer	–	–	Overflow(1)	Attempt to decrement -32,768
Negate Short Integer	–	–	Overflow(1)	Attempt to negate -32,768
Multiply Short Integer	–	–	Overflow(1)	True result is > 32,767 or < -32,768
Divide Short Integer	–	–	Domain Error Overflow(1)	Division by zero Division of -32,768 by -1
Remainder Short Integer	–	–	Domain Error	Division by zero
Equal Short Integer	–	–	None	
Not Equal Short Integer	–	–	None	
Equal Zero Short Integer	–	–	None	
Not Equal Zero Short Integer	–	–	None	
Less Than Short Integer	–	–	None	
Less Than or Equal Short Integer	–	–	None	
Positive Short Integer	–	–	None	
Negative Short Integer	–	–	None	
Range Check Short Integer	–	–	Underflow(1) Overflow(1)	< lower bound > upper bound
Convert Short Integer to Integer	–	–	None	

OPERATOR	R	P	FAULTS	COMMENT
Move Ordinal	-	-	None	
Zero Ordinal	-	-	None	
One Ordinal	-	-	None	
Save Ordinal	-	-	None	
AND Ordinal	-	-	None	
Inclusive OR Ordinal	-	-	None	
Exclusive OR Ordinal	-	-	None	
Equivalence Ordinal	-	-	None	
NOT Ordinal	-	-	None	
Extract Ordinal	-	-	None	
Insert Ordinal	-	-	None	
Significant Bit Ordinal	-	-	None	
Add Ordinal	-	-	Overflow(1)	True result is > 4,294,967,295
Subtract Ordinal	-	-	Overflow(1)	True result is < 0
Increment Ordinal	-	-	Overflow(1)	Attempt to increment 4,294,967,295
Decrement Ordinal	-	-	Overflow(1)	Attempt to decrement 0
Multiply Ordinal	-	-	Overflow(1)	True result is > 4,294,967,295
Divide Ordinal	-	-	Domain Error	Division by zero
Remainder Ordinal	-	-	Domain Error	Division by zero
Index Ordinal	-	-	None	
Equal Ordinal	-	-	None	
Not Equal Ordinal	-	-	None	
Equal Zero Ordinal	-	-	None	
Not Equal Zero Ordinal	-	-	None	
Less Than Ordinal	-	-	None	
Less Than or Equal Ordinal	-	-	None	
Convert Ordinal to Integer	-	-	Overflow(2)	Attempt to convert ordinal with non-zero high order bit
Convert Ordinal to Temporary Real	-	-	None	

OPERATOR	R	P	FAULTS	COMMENT
Move Integer	-	-	None	
Zero Integer	-	-	None	
One Integer	-	-	None	
Save Integer	-	-	None	
Add Integer	-	-	Overflow(1)	True result is > 2,147,483,647 or < -2,147,483,648
Subtract Integer	-	-	Overflow(1)	True result is > 2,147,483,647 or < -2,147,483,648
Increment Integer	-	-	Overflow(1)	Attempt to increment 2,147,483,647
Decrement Integer	-	-	Overflow(1)	Attempt to decrement -2,147,483,648
Negate Integer	-	-	Overflow(1)	Attempt to negate -2,147,483,648
Multiply Integer	-	-	Overflow(1)	True result is > 2,147,483,647 or < -2,147,483,648
Divide Integer	-	-	Domain Error Overflow(1)	Division by zero Division of -2,147,483,648 by -1
Remainder Integer	-	-	Domain Error	Division by zero
Equal Integer	-	-	None	
Not Equal Integer	-	-	None	
Equal Zero Integer	-	-	None	
Not Equal Zero Integer	-	-	None	
Less Than Integer	-	-	None	
Less Than or Equal Integer	-	-	None	
Positive Integer	-	-	None	
Negative Integer	-	-	None	
Range Check Integer	-	-	Underflow(1) Overflow(1)	< lower bound > upper bound
Convert Integer to Character	-	-	Overflow(2)	Attempt to convert integer whose value is > 255 or < 0
Convert Integer to Short Ordinal	-	-	Overflow(2)	Attempt to convert integer whose value is > 65,535 or < 0
Convert Integer to Short Integer	-	-	Overflow(2)	Attempt to convert integer whose value is > 32,767 or < -32,768
Convert Integer to Ordinal	-	-	Domain Error	Attempt to convert a negative integer value
Convert Integer to Temporary Real	-	-	None	

OPERATOR	R	P	FAULTS	COMMENT
Move Short Real	-	-	None	
Zero Short Real	-	-	None	
Save Short Real	-	-	None	
Add Short Real	x	x	Domain Error	Invalid operand
			Inexact(4)	Inexact result
Add Temporary Real to Short Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Add Short Real to Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Subtract Short Real	x	x	Domain Error	Invalid operand
			Inexact(4)	Inexact result
Subtract Temporary Real from Short Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Subtract Short Real from Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Multiply Short Real	x	x	Domain Error	Invalid operand
			Inexact(4)	Inexact result
Multiply Temporary Real by Short Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result

OPERATOR	R	P	FAULTS	COMMENT
Multiply Short Real by Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Divide Short Real	x	x	Domain Error	Invalid operand or division by zero
			Inexact(4)	Inexact result
Divide Temporary Real into Short Real	x	x	Domain Error	Invalid operand, division by zero, or division by unnormalized value
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Divide Short Real into Temporary Real	x	x	Domain Error	Invalid operand or division by zero
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Negate Short Real	-	-	Domain Error	Invalid operand
Absolute Value Short Real	-	-	Domain Error	Invalid operand
Equal Short Real	-	-	Domain Error	Invalid operand
Equal Zero Short Real	-	-	Domain Error	Invalid operand
Less Than Short Real	-	-	Domain Error	Invalid operand
Less Than or Equal Short Real	-	-	Domain Error	Invalid operand
Positive Short Real	-	-	Domain Error	Invalid operand
Negative Short Real	-	-	Domain Error	Invalid operand
Convert Short Real to Temporary Real	-	-	Domain Error	Invalid operand

OPERATOR	R	P	FAULTS	COMMENT
Move Real	-	-	None	
Zero Real	-	-	None	
Save Real	-	-	None	
Add Real	x	x	Domain Error	Invalid operand
			Inexact(4)	Inexact result
Add Temporary Real to Short Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Add Real to Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Subtract Real	x	x	Domain Error	Invalid operand
			Inexact(4)	Inexact result
Subtract Temporary Real from Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Subtract Real from Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Multiply Real	x	x	Domain Error	Invalid operand
			Inexact(4)	Inexact result
Multiply Temporary Real by Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result

OPERATOR	R	P	FAULTS	COMMENT
Multiply Real by Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Divide Real	x	x	Domain Error	Invalid operand or division by zero
			Inexact(4)	Inexact result
Divide Temporary Real into Real	x	x	Domain Error	Invalid operand, division by zero, or division by unnormalized value
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Divide Real into Temporary Real	x	x	Domain Error	Invalid operand or division by zero
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Negate Real	-	-	Domain Error	Invalid operand
Absolute Value Real	-	-	Domain Error	Invalid operand
Equal Real	-	-	Domain Error	Invalid operand
Equal Zero Real	-	-	Domain Error	Invalid operand
Less Than Real	-	-	Domain Error	Invalid operand
Less Than or Equal Real	-	-	Domain Error	Invalid operand
Positive Real	-	-	Domain Error	Invalid operand
Negative Real	-	-	Domain Error	Invalid operand
Convert Real to Temporary Real	-	-	Domain Error	Invalid operand

OPERATOR	R	P	FAULTS	COMMENT
Move Temporary Real	-	-	None	
Zero Temporary Real	-	-	None	
Save Temporary Real	-	-	None	
Add Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Subtract Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Multiply Temporary Real	x	x	Domain Error	Invalid operand
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Divide Temporary Real	x	x	Domain Error	Invalid operand, division by zero, or division by unnormalized value
			Overflow(3)	Exponent of true result is > 16,383
			Underflow(3)	Exponent of true result is < -16,382
			Inexact(4)	Inexact result
Remainder Temporary Real	-	-	Domain Error	Invalid operand, division by zero, or division by unnormalized value
			Underflow(3)	Exponent of partial result < -16383
Negate Temporary Real	-	-	Domain Error	Invalid operand
Square Root Temporary Real	x	x	Domain Error	Invalid operand, unnormalized operand, or non-zero negative operand
			Inexact(4)	Inexact result
Absolute Value Temporary Real	-	-	Domain Error	Invalid operand

OPERATOR	R	P	FAULTS	COMMENT
Equal Temporary Real	–	–	Domain Error	Invalid operand or unnormalized operand
Equal Zero Temporary Real	–	–	Domain Error	Invalid operand or unnormalized operand
Less Than Temporary Real	–	–	Domain Error	Invalid operand or unnormalized operand
Less Than or Equal Temporary Real	–	–	Domain Error	Invalid operand or unnormalized operand
Positive Temporary Real	–	–	Domain Error	Invalid operand or unnormalized operand
Negative Temporary Real	–	–	Domain Error	Invalid operand or unnormalized operand
Convert Temporary Real to Ordinal	x	–	Domain Error	Invalid operand, unnormalized operand or negative operand
			Overflow(2)	Temporary real value is > 4,294,967,295
			Inexact(4)	Inexact result
Convert Temporary Real to Integer	x	–	Domain Error	Invalid operand or unnormalized operand
			Overflow(2)	Temporary real value is > 2,147,483,647 or < -2,147,483,648
			Inexact(4)	Inexact result
Convert Temporary Real to Short Real	x	–	Domain Error	Invalid operand or unnormalized operand
			Overflow(2)	Temporary real value has exponent > 127
			Underflow(2)	Temporary real value has exponent < -126
			Inexact(4)	Inexact result
Convert Temporary Real to Real	x	–	Domain Error	Invalid operand or unnormalized operand
			Overflow(2)	Temporary real value has exponent > 1023
			Underflow(2)	Temporary real value has exponent < -1022
			Inexact(4)	Inexact result

SUB-OPERATOR FAULT GROUPS

FAULT GROUPS	TYPE	ENCODING
Store Access Descriptor Faults =>		
Level Fault	0 (FF)	01 0100
Destination Delete Rights Fault	0 (FF)	01 0011
Object Qualification Faults =>		
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Fault	0 (FF)	01 0001
Object Descriptor Type Fault	2 (TS)	0 10111
	2 (TS)	0 11111
Descriptor Allocation Faults =>		
SRO Type Rights Fault	0 (FF)	01 0111
<Object Qualification Faults (SRO)>	2 (TS)	1 01001
Object Descriptor Type Fault	2 (TS)	0 00100
Object Descriptor Exhaustion Fault	0 (FF)	01 1011
Object Descriptor Type Fault	2 (TS)	0 00000
Segment Allocation Faults =>		
<Object Qualification Faults (Claim)>	2 (TS)	1 01011
<Object Qualification Faults (PSO)>	2 (TS)	1 01010
PSO Lock Fault	0 (FF)	01 1001
Storage Block Index Overflow Fault (missing last block bit)	0 (FF)	01 1101
Storage Block Fragmentation Fault	0 (FF)	01 1110
Storage Claim Underflow Fault	0 (FF)	01 1111
Clear Memory Size Fault	0 (FF)	01 1000
Port Operation Faults =>		
<Object Qualification Faults (Carrier)>	2 (TS)	1 01000
<Object Qualification Faults (Port)>	2 (TS)	1 00111
Carrier Lock Fault	0 (FF)	01 1001
Port Lock Fault	0 (FF)	01 1010
Carrier Queued Fault	0 (FF)	01 1100
Context Qualification Faults =>		
<Object Qualification Faults (Context)>	2 (TS)	1 00100
<Object Qualification Faults (Domain)>	2 (TS)	1 00010
<Object Qualification Faults (Instruction)>	2 (TS)	1 00011
Process Binding and Qualification Faults =>		
<Object Qualification Faults (Process)>	2 (TS)	1 00101
Process Level Object Lock Fault	0 (FF)	11 0001
<Context Qualification Faults>		

NON-INSTRUCTION INTERFACE FAULTS

OPERATOR	TYPE	ENCODING
Initialization =>		
<Object Qualification Faults (Processor)>	2 (TS)	1 00110
<Object Qualification Faults (Obj. Table Directory)>	2 (TS)	1 00001
<IPC Faults>		
IPC Faults =>		
<Object Qualification Faults (PCO)>	2 (TS)	1 10000
PCO Response Count Fault	0 (FF)	11 0010
PCO Lock Fault	0 (FF)	11 0011
Idle =>		
<Delay Port Service Faults>		
Process Binding =>		
<Object Qualification Faults (Carrier)>	2 (TS)	1 01000
Process Lock Fault	0 (FF)	11 0001
<Process Qualification Faults>		
<Port Operation Faults>		
Process Selection =>		
<Delay Port Service Faults>		
<Object Qualification Faults (Carrier)>	2 (TS)	1 01000
<Port Operation Faults>		

OBJECT OPERATOR FAULTS

OPERATOR	TYPE	ENCODING
Branch		
Branch True		
Branch False		
Instruction Pointer Overflow Fault	1 (IP)	
Instruction Object Displacement Fault	0 (FF)	10 0000
Branch Indirect		
Instruction Object Displacement Fault	0 (FF)	10 0000
Branch Intersegment		
Branch Intersegment without Trace		
Branch Intersegment and Link		
<Object Qualification Faults (Instruction)>	2 (TS)	1 00011
Instruction Object Displacement Fault	0 (FF)	10 0000
Breakpoint		
no explicit fault cases		
Copy Access Descriptor		
<Store Access Descriptor Faults>		
Null Access Descriptor		
Destination Delete Rights Fault	0 (FF)	01 0011
Amplify Rights		
TCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (TCO)>	2 (TS)	1 01110
Type Fault	0 (FF)	01 1000
Race Condition Fault (the access descriptor was changed before the amplified value is stored back)	0 (FF)	01 0011
Restrict Rights		
no explicit fault cases		
Retrieve Type Definition		
Source AD Validity Fault	0 (FF)	01 0101
<Store Access Descriptor Faults>		

OPERATOR	TYPE	ENCODING
Create Refinement		
Source AD Validity Fault	0 (FF)	01 0101
Object Descriptor Type Fault	2 (TS)	0 00110
Offset and Length Compatibility Fault	0 (FF)	01 1001
Refinement Overflow Fault	0 (FF)	01 1010
<Descriptor Allocation Faults>		
Level Fault	0 (FF)	01 0100
<Store Access Descriptor Faults>		
Create Typed Refinement		
TCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (TCO)>	2 (TS)	1 01110
Source AD Validity Fault	0 (FF)	01 0101
Object Descriptor Type Fault	2 (TS)	0 00110
Type Fault	0 (FF)	01 1000
Offset and Length Compatibility Fault	0 (FF)	01 1001
Refinement Overflow Fault	0 (FF)	01 1010
<Descriptor Allocation Faults>		
Level Fault	0 (FF)	01 0100
<Store Access Descriptor Faults>		
Create Object		
<Descriptor Allocation Faults>		
<Segment Allocation Faults>		
<Store Access Descriptor Faults>		
Create Typed Object		
TCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (TCO)>	2 (TS)	1 01110
<Descriptor Allocation Faults>		
Level Fault	0 (FF)	01 0100
<Segment Allocation Faults>		
<Store Access Descriptor Faults>		
Inspect Access Descriptor		
no explicit cases		
Inspect Object		
Access Path Object Descriptor Validity Fault	0 (FF)	01 0101
Equal Access		
Move to Embedded Data Value		
Move from Embedded Data Value		
no explicit fault cases		
Lock Object		
Source Representation Rights Fault	0 (FF)	01 0110
<Object Qualification Faults>		
Unlock Object		
Source Representation Rights Fault	0 (FF)	01 0110
<Object Qualification Faults>		
Object Lock ID/Type Fault	0 (FF)	01 1001

OPERATOR	TYPE	ENCODING
Indivisibly Add Short Ordinal		
Indivisibly Add Ordinal		
Indivisibly Insert Short Ordinal		
Indivisibly Insert Ordinal		
no explicit fault cases		
Enter Environment 1, 2, 3		
<Object Qualification Faults>		
Copy Process Globals		
<Store Access Descriptor Faults>		
Set Context Mode		
no explicit fault cases		
Adjust Stack Pointer		
no explicit fault cases		
Call		
Call through Domain		
<Object Qualification Faults (Domain)>	2 (TS)	1 00010
Domain Access Index Overflow Fault	0 (FF)	01 0010
Instruction Object Type Rights Fault	0 (FF)	01 0101
<Object Qualification Faults (Instruction)>	2 (TS)	1 00011
Context Parameters Size Fault	0 (FF)	01 0110
Context Type Rights Fault	0 (FF)	01 0111
<Object Qualification Faults (Context)>	2 (TS)	1 00100
Instruction Object Displacement Fault	0 (FF)	10 0000
Return		
Context Type Rights Fault	0 (FF)	01 0111
<Context Qualification Faults>		
<Object Qualification Faults (PSO)>	2 (TS)	1 01010
<Object Qualification Faults (Object Table)>	2 (TS)	0 00100
PSO Lock Fault	0 (FF)	01 1001
Instruction Object Displacement Fault	0 (FF)	10 0000
Return and Fault		
<Return>		
Return Fault	0 (FF)	00 0100
Send		
Receive		
Conditional Send		
Conditional Receive		
Surrogate Send		
Surrogate Receive		
Surrogate Carrier Validity Fault	0 (FF)	01 0101
Surrogate Carrier Type Rights Fault	0 (FF)	01 0101
Destination Port Type Rights Fault	0 (FF)	01 0111
Port Type Rights Fault	0 (FF)	01 0110
Level Fault	0 (FF)	01 0100
<Port Operation Faults>		

OPERATOR	TYPE	ENCODING
Delay Process		
Send Process		
Port Type Rights Fault	0 (FF)	01 0110
Level Fault	0 (FF)	01 0100
<Port Operation Faults>		
Set Process Mode		
Process Object Type Rights Fault	0 (FF)	01 0110
Process Object Access Mismatch Fault	0 (FF)	01 1000
Read Process Clock		
no explicit fault cases		
Send to Processor		
PCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (PCO)>	2 (TS)	1 01010
Read Processor Status		
no explicit fault cases		
Move to Interconnect		
Move from Interconnect		
Odd Displacement Fault	0 (FF)	01 0110
Odd Interconnect Descriptor Base Address Fault	0 (FF)	01 0101
<Object Qualification Faults (Interconnect)>	2 (TS)	0 01100
Block Move		
Offset Overflow	0 (FF)	00 0001

TRACE REFERENCE

TRACE OPERATION

GDP support for software debugging and analysis is based on the tracing mechanism. When a trace event occurs, the instruction object Domain Access Index (DAI), the instruction pointer, and a trace code are recorded in the Trace Control Data Area in the current context data part. An intersegment branch is then effectively executed to bit displacement 64 of the Trace Instruction Object (specified by AD 1 in the current defining domain).

A process may be in one of four trace modes as determined by the setting of the Trace Mode field in the current process status:

00 - No Trace Mode (not tracing)
01 - Fault Trace Mode
10 - Flow Trace Mode
11 - Full Trace Mode

These trace modes are defined as follows:

No Trace Mode: Process execution is as described throughout the rest of this manual.

Fault Trace Mode: A trace event occurs prior to the execution of the first instruction of a context-level fault handler.

Flow Trace Mode: A trace event occurs prior to the execution of the first instruction of a context-level fault handler. A trace event occurs after the execution of all instructions that change the current instruction object. This includes all intersegment branch instructions, CALL, CALL THROUGH DOMAIN, RETURN, and RETURN AND FAULT instructions. In RETURN or RETURN AND FAULT instructions, a trace event <u>also</u> occurs prior to the execution of the instruction.

Full Trace Mode: A trace event occurs prior to the execution of the first instruction of a context-level fault handler. A trace event occurs prior to the execution of every instruction.

When a trace mode other than no trace is specified by a process, trace events are generated as described whenever the process is executing an instruction object which is opened for tracing. An instruction object is open for tracing if it is accessed via an AD which has trace rights. If an instruction object is not open for tracing to the process, then no trace events are ever generated from within it.

When a process which is in full trace mode transfers from a closed instruction object to an open one, the first trace event generated is prior to the execution of the first instruction in the open instruction object. When such a process transfers from an open instruction object to a closed one, the last trace event is generated immediately prior to the instruction which causes the transfer.

When a process is in flow trace mode, the first trace event generated is prior to the execution of the first instruction in any open instruction object independent of whether the transfer instruction is located in an open or closed one. When a process is in the flow trace mode, a trace event is also generated prior to the execution of a Return instruction if the current instruction object is an open one. This allows the current context and its associated local objects to be examined before they are reclaimed by the Return instruction. If the instruction object that a context returns to is an open one, another trace event is generated prior to the execution of the next instruction after the CALL or CALL THROUGH DOMAIN instruction.

Access descriptors to trace instruction objects, and to any associated instruction objects (i.e., objects that the trace routine can branch to) should have their Trace Rights bit Off. Otherwise, it may be possible to get into an infinite loop. The current implementation always forces the processor's trace enable off when branching to the trace routine.

The trace mode bits of the current process can be changed by the SET PROCESS MODE instruction.

TRACE CONTROL DATA AREA

The Trace Control Data Area is located in the context data part and is organized as follows:

	Byte Displacement
Trace Event Code	12
Trace Instruction Pointer	10
Trace DAI	8

These fields have the following meaning:

Trace DAI (Bytes 8 - 9)
This 16-bit field records the DAI in the defining domain of the instruction object in which the trace event should resume.

Trace Instruction Pointer (Bytes 10 - 11)
This 16-bit field records the instruction pointer of the instruction with which execution can resume after the trace event.

Trace Event Code (Bytes 12 - 13)
This 16-bit field records the encoding for the method used to resume. The trace event code is defined as follows:

1 indicates that normal instruction flow can be resumed by a BRANCH INTERSEGMENT WITHOUT TRACE instruction using the Trace DAI and the Trace Instruction Pointer.

2 indicates that normal instruction flow has to be resumed by executing a RETURN instruction.

3 indicates that normal instruction flow has to be resumed by executing a RETURN AND FAULT instruction.

4 indicates fault trace and no Trace DAI nor Trace Instruction Pointer is recorded.

5 indicates a BREAKPOINT instruction is executed.

GLOSSARY

This glossary defines important terms used in this manual. Within the definitions, references to other terms defined in this glossary are underscored.

List of Terms Defined

access descriptor
access environment
access part
access rights
access selector
AD rights
attached processor

bit-field specifier
blocked
Boolean

carrier
central system
character
compaction
context object

data part
defining domain
delete rights
domain
domain access index
dynamic type
dynamic-type object

embedded data value

fault
FIFO
forwarding
fragmentation

garbage collection
general data processor
generic object
global heap SRO

heap SRO

instruction object
integer
interconnect
interface processor
interprocessor communication

level
level check
lifetime strategy
LIFO
local heap SRO

message

object
object descriptor
object lock
object reference
object table
object table directory
object type
operand stack
ordinal

package
peripheral subsystem
physical storage object
port
process
process globals object
processor communication object
processor object
processor type

read rights
real
refinement
representation rights

segment
short integer
short ordinal
short real
stack SRO
storage claim object
storage resource object
system object
system type

temporary real
type
type control object
type definition object
type manager
type rights

unchecked copy rights

write rights

access descriptor (AD): a reference to an iAPX 432 object that restricts operations on both the object using the AD (access rights) and on the AD itself (AD rights). The iAPX 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 iAPX 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 object of the process that contains the context, and by any access descriptors passed as parameters from its caller, returned as results from operations called by the context, received in interprocess communication, or created during the context's execution.

access part: a distinct, optional part of an iAPX 432 object that can only contain access descriptors (ADs). The hardware limits operations on object access parts to ensure that ADs are not corrupted. An access part can contain from 0 to 16,384 ADs.

access rights: attributes of an iAPX 432 access descriptor (AD) that restrict operations on the referenced object using the AD. Access rights consist of representation 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 selector: a 16-bit data value that selects an access descriptor (AD) from the current access environment of a context. The ENV selector field in an access selector selects one of up to four objects that determine the current access environment. The access index field in an access selector selects an AD from the access part of the object specified by the ENV selector.

AD rights: attributes of an iAPX 432 access descriptor (AD) that restrict operations on the AD itself. AD rights consist of delete rights and unchecked copy rights.

attached processor (AP): a processor, usually an Intel microprocessor, which controls the peripheral subsystem. The peripheral subsystem contains peripheral devices, controllers, memory, the AP, and an iAPX 432 interface processor (IP), all communicating on the peripheral subsystem's bus. Software running on the AP controls the IP.

bit-field specifier: specifies a field of bits within an ordinal or short-ordinal operand. The specifier gives the beginning bit position and width in bits of the field.

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.

Boolean: a one-byte value of the character data type, used to represent logical TRUE (xxxxxxx1) or FALSE (xxxxxxx0).

carrier: an iAPX 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 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 iAPX 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 are provided by one or more peripheral subsystems. The iAPX 432 interface processors are part of both systems and provide the bridge between them.

character: a one-byte computational data type used to represent text characters, Booleans, and unsigned integers in the range 0 to 255.

compaction: an operating system 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).

context object: an iAPX 432 system object that represents an activation of a subprogram.

data part: a distinct, optional part of an iAPX 432 object that can contain any information except access descriptors (ADs). Programs with proper rights to an object can make arbitrary changes to the object's data part using any of the iAPX 432 data operators. A data part can contain from 0 to 65536 bytes.

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 an iAPX 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.

domain: an iAPX 432 system object that represents a program module and can contain or reference multiple subprograms and data elements. A domain is a major part of the access environment of a 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.

domain access index: a 16-bit data value that selects an access descriptor (AD) from the defining domain of the current context. The low two bits are ignored; the high 14 bits index into the domain access part.

dynamic type: an iAPX 432 object type defined by software and represented by an iAPX 432 type definition object (TDO).

dynamic-type object: an instance of a dynamic type.

embedded data value (EDV): a 31-bit ordinal value stored (embedded) in a null access descriptor. Special operations are provided to convert ordinals to and from EDVs. EDVs allow simple messages to be sent between processes in null ADs without the overhead of allocating and referencing a separate message object.

fault: a processor-detected error during program execution. For example, if an addition operation overflows, the GDP detects the error and raises a fault. There are three levels of faults, increasing in severity: context faults, process faults, and processor faults.

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 iAPX 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 noncontiguous blocks, caused by the normal operation of heap allocation and garbage collection.

garbage collection: a concurrent operating system process that detects unreferenced objects and reclaims the corresponding descriptors and storage. Garbage collection runs concurrently with user processes and is invisible to them.

general data processor (GDP): the main type of processor provided by Intel to execute within the iAPX 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 object: an iAPX 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, dynamic-type objects, or interconnect objects.

global heap SRO: a heap SRO at level zero; only garbage collection can reclaim storage allocated from a global heap.

heap SRO: an iAPX 432 storage resource object (SRO) 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.

instruction object: an iAPX 432 system object that contains GDP instructions, typically for a single subprogram.

integer: a four-byte computational data type used to represent signed whole numbers in the range -2,147,483,648 to 2,147,483,647.

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 objects which are normally defined at system initialization.

interface processor (IP): an iAPX 432 processor that connects an iAPX 432 central system to one peripheral subsystem. 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 iAPX 432 central system.

interprocessor communication (IPC): a protocol for sending special interprocessor message codes (IPCs) between iAPX 432 processors.

level: a short-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 one 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 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 (AD) 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 SRO, local heap SRO, and stack SRO.

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.

message: any iAPX 432 object for which an access descriptor (AD) is copied from a sending process to a receiving process.

object: a data structure within memory described by an object descriptor and accessed via access descriptors (ADs). Objects are the iAPX 432 construct for access control, run-time type checking, storage management, and program addressing.

object descriptor: an object table enty that gives object attributes needed by the iAPX 432 processors, e.g., type and storage information.

object lock: a double-byte field in many iAPX 432 system objects used to synchronize concurrent hardware and/or software access to the object containing the object lock. Properly used, an object lock ensures that a process or processor has exclusive access to an object while reading or updating it.

object reference: see access descriptor.

object table: an iAPX 432 system object containing object descriptors.

object table directory (OTD): a special object table that only contains object descriptors for all other object tables in an iAPX 432 system.

object type: type information given in object descriptors for storage segments and refinements, consisting of system type and processor type.

operand stack: an area within a context data part that provides an expression evaluation stack for the context.

ordinal: a four-byte computational data type used to represent unsigned integers in the range 0 to 4,294,967,295, and also to represent bit strings of 32 bits or less.

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 specification. The package body, which can be separately compiled, contains the bodies (implementations) of subprograms, tasks, or other packages declared in the package specification. A package is represented by an iAPX 432 domain.

peripheral subsystem: a computer system controlled by an attached processor (AP), which manages one or more peripheral devices and is linked to an iAPX 432 central system by an iAPX 432 interface processor (IP). The peripheral subsystem contains peripheral devices, controllers, memory, the AP, and an IP.

physical storage object (PSO): an iAPX 432 system object that provides a free storage pool for use by an iAPX 432 storage resource object (SRO).

port: an iAPX 432 system object that provides a queuing mechanism with two queues, a bounded message queue and an unbounded carrier queue. Ports support FIFO, priority, and deadline-within-priority queuing. Ports are used for interprocess communication and process scheduling and dispatching.

process: an iAPX 432 system object that represents part of a program that execute concurrently with other parts, also represented as processes. Because processes can compete for execution time, scheduling information is associated with processes for use in selecting the next process to run, and to ensure that a process does not monopolize a processor for longer than some time limit. A program can consist of one or more processes.

process globals object (PGO): an iAPX 432 generic object that is designated by a process as its globals object and that provides access to additional attributes of a process's run-time environment. For example, a PGO can reference a heap SRO used by the process to allocate heap objects.

processor communication object (PCO): an iAPX 432 system object used to send and receive interprocessor messages (IPCs).

processor object: an iAPX 432 system object that contains state information for one physical iAPX 432 processor.

processor type: an iAPX 432 object type field; each of its values designates the kinds of processors that can reference objects with that processor type. The alternatives are GDP only, IP only, and "all" (both GDP and IP).

read rights: attribute of an iAPX 432 access descriptor (AD) that controls the right to read the referenced object.

real: an eight-byte computational data type used to represent signed floating point numbers with magnitudes in the range $2.2*10^{-308}$ to $1.8*10^{308}$ (and also zero).

refinement: an iAPX 432 object that is contained within another object. When a refinement is created, the displacement of the base of the refinement in the underlying object and the size of the refinement are specified and are checked by the GDP.

representation rights: attributes of an iAPX 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 locations, from 0 to 131,072 (128K) bytes in apparent size, defined by an iAPX 432 object descriptor. A segment can be in the storage address space or the interconnect address space. (Interconnect segments are a maximum of 65,536 bytes.)

short integer: a two-byte computational data type used to represent signed whole numbers in the range -32,768 to 32,767.

short ordinal: a two-byte computational data type used to represent unsigned integers in the range 0 to 65,535, and also to represent bit strings of 16 bits or less.

short real: a four-byte computational data type used to represent signed floating point numbers with magnitudes in the range $1.2*10^{-38}$ to $3.4*10^{38}$ (and also zero).

stack SRO: an iAPX 432 storage resource object (SRO) built in to the process object of a process and used for allocation of objects with lifetimes local to the creating context. Allocation and deallocation for a stack SRO are strictly LIFO (Last-In-First-Out). There is no object descriptor for a stack SRO, but it is conceptually a distinct object.

storage claim object (SCO): an iAPX 432 system object used to limit the total number of bytes of physical storage allocated via the set of heap SROs that reference the SCO.

storage resource object (SRO): an iAPX 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 of objects allocated from it. A heap SRO can also specify a storage claim object that limits the amount of physical storage that can be allocated from a particular set of SROs.

system object: an iAPX 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 dynamic type objects.

system type: an iAPX 432 object type field that distinguishes a class of iAPX 432 objects with a particular processor-recognized meaning.

temporary real: a ten-byte computational data type used to represent signed floating point numbers with magnitudes in the range $1.7*10^{-4932}$ to $1.2*10^{4932}$ (and also zero).

type: a set of values with certain operations and representations defined for the set.

type control object (TCO): an iAPX 432 system object that provides the right to amplify specific rights on access descriptors (ADs) for objects of a particular type, and/or the right to create objects of the type, and/or the right to create refinements of the type.

type definition object (TDO): an iAPX 432 system object that represents a particular software-defined dynamic type.

type manager: an Ada package or iAPX 432 domain object that defines all basic operations on a certain type of object. Any other operations on objects of the type must be composed by using the basic operations. A type manager may distribute accesses for the managed objects, but normally retains for itself the rights to directly read or write those objects.

type rights: attributes of an iAPX 432 access descriptor (AD) that restrict the right to execute certain operations using the AD, depending on the type of object it references. For example, create rights are required to create a new object using an SRO access.

unchecked copy rights: an attribute of an iAPX 432 access descriptor (AD) which, if set, 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 (i.e., if a level check could never fail). Amplifying unchecked copy rights is a privileged operation, used only by O.S. software.

write rights: an attribute of an iAPX 432 access descriptor that controls the right to write the referenced object.

intel®

iAPX 432 General Data Processor
Architecture Reference Manual
171860-004

REQUEST FOR READER'S COMMENTS

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

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

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

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

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

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

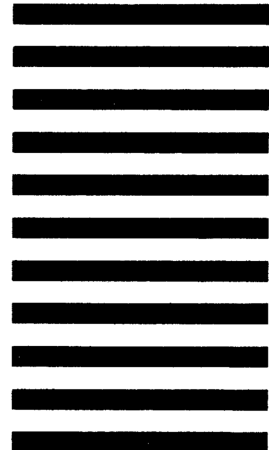
5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ____________

NAME ________________________ DATE __________
TITLE ________________________
COMPANY NAME/DEPARTMENT ________________________
ADDRESS ________________________
CITY ______________ STATE ______________ ZIP CODE __________
(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS...

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



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 NE Elam Young Parkway
Hillsboro OR 97123

ISO-N TECHNICAL PUBLICATIONS

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

IS-055/1K/0284/OSPS
MICROPROCESSORS