National Computer Security Center **NCSC-FER-95/006.C**

Report No. 95/006.C
Library No. S-244,633

FINAL EVALUATION

International Business Machines Corporation

Application System/400

V4R1

October 19, 1998

National Security Agency
9800 Savage Road
Fort George G. Meade, Maryland 20755-6000

Trademarks

- IBM® is a registered trademark of the International Business Machines Corporation.
- APPN® is a registered trademark of the International Business Machines Corporation.
- Application System/400® is a registered trademark of the International Business Machine Corporation.
- AS/400® is a registered trademark of the International Business Machine Corporation.
- Operating System/400® is a registered trademark of the International Business Machine Corporation.
- OS/400® is a registered trademark of the International Business Machine Corporation.
- C/400® is a registered trademark of the International Business Machine Corporation.
- GDDM® is a registered trademark of the International Business Machine Corporation.
- Operational Assistant® is a registered trademark of the International Business Machine Corporation.
- RT® is a registered trademark of the International Business Machine Corporation.
- SAA® is a registered trademark of the International Business Machine Corporation.
- SQL/400® is a registered trademark of the International Business Machine Corporation.
- 400® is a registered trademark of the International Business Machine Corporation.
- HP® is a registered trademark of the Hewlett-Packard Company.
- UNIX® is a registered trademark of X/Open Company Limited.
- Advanced Function Printing™ is a trademark of the International Business Machine Corporation.
- AFP™ is a trademark of the International Business Machine Corporation.
- Integrated Language Environment™ is a trademark of the International Business Machine Corporation.
- ILE™ is a trademark of the International Business Machine Corporation.
- OfficeVision/400™ is a trademark of the International Business Machine Corporation.
- PrintManager™ is a trademark of the International Business Machine Corporation.
- PSF™ is a trademark of the International Business Machine Corporation.
- RM/COBOL™ is a trademark of the Ryan McFarland Corporation.
- System/36™ is a trademark of the International Business Machine Corporation.

- System/38™ is a trademark of the International Business Machine Corporation.
- Motorola™ is a trademark of Motorola, Inc..

All other brands and product names are trademarks or registered trademarks of their respective companies.

Foreward

This publication, the *International Business Machines Corporation Application System/400 Final Evaluation*, is being issued by the National Computer Security Center under the authority of and in accordance with DoD Directive 5215.1, “Computer Security Evaluation Center.” The purpose of this report is to establish the rating for the Application System/400 product from International Business Machines Corporation. The requirements stated in this report are taken from the *Department of Defense Trusted Computer System Evaluation Criteria* dated December 1985.

Approved:

John C. Davis October 19, 1998

Director, National Computer Security Center

Acknowledgments

The Security Analysis (SA) team for the Application System/400 product, V4R1, from International Business Machines Corporation consisted of the following individuals, who were affiliated with the indicated organizations:

Mike Allen

The Aerospace Corporation
Columbia, MD

Victoria A. Ashby

The MITRE Corporation
McLean, VA

T. Ronald Chrisope
Edward A. Fishel
John R. Griffith
Glen W. Nelson
David O. Lewis
Richard R. Sanders

International Business Machines Corporation
Rochester, MN

John F. Walsh

National Security Agency
Fort George G. Meade, MD

This evaluation is based on the evaluation Application System/400 product, V2R3, which was performed by the following individuals, who were affiliated with the indicated organizations:

James J. Donndelinger
Jandria S. Alexander
Ramzi A. Haraty
Scott A. Ryden

The Aerospace Corporation
Columbia, MD

Victoria A. Ashby
Sarah M. Weinberg

The MITRE Corporation
McLean, VA

Anthony J. Nadalin

International Business Machines Corporation
Rochester, MN

Capt. Joseph C. Pulido, USAF
John F. Walsh

National Security Agency
Fort George G. Meade, MD

Table of Contents

Section 5. Trusted Computing Base (TCB) Protected Resources 5-1

List of Figures

List of Tables

Executive Summary

The security protection provided by the International Business Machines Corporation Application System/400 (AS/400) product, configured in a secure manner as described in the Trusted Facilities Manual, running on any of the platforms in the evaluated hardware configuration, has been examined by a National Security Agency (NSA) product evaluation team on behalf of the National Computer Security Center (NCSC). The security features of AS/400 were examined against the requirements specified by the Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC), December 1985 [2], to establish a rating.

The NSA product evaluation team has determined that the highest rating for which the Application System/400 satisfies the specified requirements of the applicable criteria is C2.

In July of 1991, an effective Memorandum of Understanding was entered into by the International Business Machines Corporation (IBM) and the National Computer Security Center (NCSC) for the NCSC to evaluate Version 2, Release 2 (V2R2) of the IBM Application System/400 Operating System (OS/400) running on IBM's AS/400 hardware platforms. The evaluation effort started in July 1992 on Version 2, Release 3 (V2R3). AS/400 was examined against the requirements specified by the Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC), December 1985 [2]. The evaluated IBM AS/400 was configured according to its Trusted Facilities Manual (TFM).

AS/400 was submitted as a candidate for a C2 class rating, and the NCSC determined that C2 is, in fact, the highest class at which AS/400 satisfies all the requirements of the TCSEC. AS/400 is a family of midrange computers based on a single software architecture using OS/400. OS/400 is a layered-design operating system, developed using object-based techniques. This design was intended to keep the operating system isolated from any changes at the hardware level. As a result of the object-based design techniques, IBM provides a consistent access control mechanism across all system objects. OS/400 is implemented on a proprietary hardware base developed by IBM. The hardware base includes intelligent controllers and supports a wide array of peripherals.

Successful evaluation at the C2 level of trust indicates that AS/400 provides a Trusted Computing Base (TCB) that provides identification and authentication, discretionary (need-to-know) protection, and, through the inclusion of audit capabilities, for accountability of subjects and the actions they initiate. The IBM AS/400, V2R3, became available to customers in December of 1993. The evaluated version of AS/400, V2R3 feature code 1920, became available to customers in September of 1995. IBM is following approved Rating Maintenance Phase (RAMP) procedures to quickly RAMP to the most current AS/400 product version available soon after completing Final Evaluation of the base AS/400 product.

In October of 1996, IBM satisfied the RAMP procedures for the V3R0.5 version of the AS/400 TCB, running on a new set of hardware known as the Advanced Series, to the C2 level of trust. The V3R0.5 and Advanced series were made available to customers in June of 1994. The Advanced Series hardware is functionally equivalent to the previously evaluated hardware. The software was primarily a fix release with no significant security relevant changes.

In October of 1997, IBM satisfied the RAMP procedures for the V3R2 version of the AS/400 Software TCB to the C2 level of trust. The hardware portion of the TCB remained the same as the V3R0.5 evaluated configuration. The software was enhanced to support the porting of POSIX applications, including a hierarchical file system, an execute bit, and multiple group capability. The V3R2 version was made available to customers in July of 1996.

In October of 1998, IBM, with System Analysis Team support, satisfied the RAMP procedures for the V4R1 version of the AS/400 TCB, running on a new set of hardware that uses Reduced Instruction Set Computer (RISC) technology, to the C2 level of trust. The software was primarily updated to support the new hardware. The V4R1 version was made available to customers in August of 1997.

Section 1
Introduction

In July 1992, the National Computer Security Center (NCSC) began evaluation of Application System/400 from International Business Machines Corporation. This report summarizes the results of that evaluation, and provides evidence and analysis of the security features and assurance characteristics provided by AS/400. Material for this report was gathered by the evaluation team through analysis of documentation and interaction with product developers. This report documents the evaluation team's understanding of the product's security design and appraises it against the C2 requirements as defined by the Department of Defense (DoD) Trusted Computer System Evaluation Center (TCSEC), December 1985 [2].

This evaluation applies to the version of AS/400 specified by Appendix A, "Evaluated Hardware", on page A-1 and Appendix B, "Evaluated Software", on page B-1.

1.1. Evaluation Process Overview

The National Security Agency (NSA) evaluation process has three phrases: Pre-evaluation, Evaluation, and Ratings Maintenance. Pre-evaluation begins when the vendor submits a product proposal for review. The proposal is reviewed in relation to market potential, security design, and state-of-the-art technology. Products that are deemed appropriate candidates for evaluation proceed to the next step in pre-evaluation: Technical Assessment (TA). The purpose of the TA is to determine the degree of advice assistance that might be necessary before the product is ready to begin evaluation. The TA team is composed of experienced evaluators who provide a quick assessment of the vendor's readiness for evaluation. Depending upon the results of the TA, a vendor will receive advice or complete an Intensive Preliminary Technical Review (IPTR). During an IPTR, the vendor demonstrates that the product design and evaluation evidence are complete. An IPTR, if successful, allows NSA to initiate a product evaluation.

Evaluation is an analysis of the trusted components of the product. During the evaluation phase, a team undergoes training on the product, performs security analysis, conducts hands-on activities appropriate to the products rating (such as system testing), and reviews design, user, administrator, test, and ratings maintenance documents. The evaluation team uses the documentation to produce an Initial Product Assessment Report (IPAR). This document, along with a team presentation of the security architecture of the product and the team's plans for testing, are reviewed by a Technical Review Board (TRB), which consists of senior members of the evaluation community. The TRB's purpose is to ensure consistency of analysis across all evaluations. The TRB makes a recommendation to NSA management, which issues an Evaluation Milestone Decision (EMD) detailing the actions the team must complete in order to proceed. Once these actions have been addressed, the team proceeds to perform security testing and other rating-appropriate activities (e.g., architecture studies) on the product. The IPAR is updated with the results of these activities and post-IPAR security analysis to produce the Final Evaluation Report (FER). The evaluation team presents the FER, along with the final results of testing, to the TRB for review. The TRB reviews the results of testing, and makes a recommendation to NSA management. NSA management reviews the TRB recommendation and issues a Final EMD, detailing any actions required to complete the evaluation. After the team addresses the actions in the final EMD, the product is placed on the Evaluated Products List (EPL) and the FER is published.

After completion of the initial evaluation, the product enters the Ratings Maintenance Phase (RAMP). RAMP provides a mechanism for a vendor to maintain the rating of a product throughout its life cycle. During RAMP, the vendor works with an NSA Technical Point-of-Contact (TPOC) to discuss proposed changes to the evaluated product. The Vendor Security Analyst (VSA), possibly in conjunction with an NSA-assigned Security Analysis team (SA-Team), performs security analysis of the product. All security-relevant changes are presented to a TRB, which recommends to NSA management whether the updated product should retain its rating. Once all changes are approved, an entry is made on the EPL, and the FER is updated by the vendor.

1.2. Document Organization

Section 2, “Product Overview” on page 2-1 presents a high-level overview and a brief history of the Application System/400. A definition of the TCB for the Application System/400 is also located in this section.

Section 3, “Machine Architecture” on page 3-1 provides a description of the Application System/400 machine. This description is broken into two major discussions: the hardware and the System Licensed Internal Code (SLIC).

Section 4, “Software Architecture” on page 4-1 provides a description of the Operating System/400 (OS/400) and the other software that is under evaluation. This description is broken into three major discussions: the base operating system, the integrated data base, and major support functions such as Initial Program Load (IPL) and software installation.

Section 5, “Trusted Computing Base (TCB) Protected Resources” on page 5-1 provides a description of the resources protected by the Application System/400 TCB.

Section 6, “TCB Protection Policies” on page 6-1 provides a description of the TCB protection policies of the Application System/400. The discussion includes policies for Identification and Authentication, Discretionary Access Control, Special Authority, Object Reuse, and Audit.

Section 7, “Assurance” on page 7-1 provides a description of the Application System/400 features which provide assurance that the TCB policies are correctly implemented.

Section 8, “Evaluation of Application System/400 Against the Applicable Requirements” on page 8-1 provides a description of the requirements for a C2 rating and a description of how the Application System/400 satisfies each requirement.

Section 9, “Features and Assurance Beyond The Evaluated Rating” on page 9-1 provides a description of all requirements beyond C2 that the Application System/400 satisfies.

Section 10, “Evaluator Comments” on page 10-1 provides comments from the evaluation team regarding the Application System/400.

Appendix A, “Evaluated Hardware”, on page A-1 provides a listing of the evaluated hardware configuration.

Appendix B, “Evaluated Software”, on page B-1 provides a listing of the evaluated software configuration. This includes SLIC, OS/400, and applicable Licensed Programs.

Appendix C, “Draft Product Bulletin/EPL Entry”, on page C-1 provides a draft EPL entry.

Appendix D, “Relevant Decisions”, on page D-1 provides a record of any posted decisions or interpretations that are relevant to the evaluation of the Application System/400.

Appendix E, “Acronyms and Nomenclature”, on page E-1 provides a list of acronyms and nomenclature used in this document.

Appendix F, “References and Sources”, on page F-1 provides a list of references to other documents used as source material for this document.

Appendix G, “Security Related System Values”, on page G-1 provides a list of the security relevant system values and their required settings.

Appendix H, “System Jobs”, on page H-1 provides a list of the types of jobs started by the system to provide system services.

Appendix I, “IBM-Supplied User Profiles”, on page I-1 provides a list of user profiles that are supplied by IBM.

1.3. Conventions

Throughout this FER, specific conventions are used in order to keep the flow concise, clear, and consistent. The following list describes the conventions used:

- References to “user applications” include licensed programs
- References to “the machine” indicate the working collection of components below the Machine Interface (MI)
- References to “devices” indicate non-programmable devices that are not dumb (i.e., devices that can respond to input/output processors).
- References to “user profile” should be taken to mean “user profile” or “group profile”
- “Workstation” should be taken to mean a display device from which to take work, and does not imply that the display device possesses any intelligence
- Whenever the term ‘object’ is used, it is preceded by the type of object referred to(i.e., MI object, internal object, or external object)

Instructions and commands referred to in this FER also follow conventions. If an instruction or command is referred to only one time, then the text refers to the instruction/command generically (e.g., an MI instruction to create a space...). However, if the instruction/command is referred to more than once, the conventions listed below are followed.

- CL commands are written with the phrase “CL command” preceding the name of the command in capital letters (e.g., the CL command PWRDWNSYS is used to...).
- APIs are written with the phrase “API program” preceding the name of the program (e.g., the API program QDMRTVFO...).
- MI instructions are written with the phrase “MI instruction” preceding the English name of the instruction (e.g., the MI instruction Create Space Object...).
- RISC instructions are written with the phrase “RISC instruction” preceding the name of the instruction (e.g., the RISC instruction ADD...).

Section 2
Product Overview

This section provides a brief product history and an introduction to IBM AS/400 Version 4 Release 1 (V4R1), running on Version 3 or 4 Advanced Series Reduced Instruction Set Computer (RISC) hardware. It begins with a brief history of AS/400, followed by a high-level overview of the AS/400 system as a whole. The layered architecture is introduced in this overview, as is how the layered architecture supports an abstract machine. Next, some terms are defined to aid understanding. In addition to the layered architecture, AS/400 has six distinguishing features: it uses state and domain concepts, it is object-based, it has an integrated database, it uses a single-level store, it provides hardware storage protection, and it has tagged pointers. A brief introduction to each of these distinguishing features is given. Next, the security features provided are discussed. After this, the Trusted Computing Base (TCB) of AS/400 is introduced.

2.1. Product History

The International Business Machines (IBM) Corporation’s Application System/400 (AS/400) is a family of midsized computers designed for common business applications such as billing, inventory control, accounts receivable, and sales analysis. More than 400,000 AS/400 systems have been installed worldwide.

As a new generation of the IBM System/3x family, the AS/400 follows the System/3, System/32, System/34, System/36, and System/38. These systems have been produced at IBM’s site in Rochester, Minnesota, since the late 1960s. The mission of these systems was to bring cost-effective, ease-of-use computing to the small and medium-size business and government entities who were looking for turnkey computing. During the 1980s, IBM had two offerings: the System/36, announced in 1983, which was essentially a follow-on to the System/3, 32, and 34; and the System/38, announced in 1978, which was developed as a totally new operating system based on mid-1970s technology.

In 1988, the AS/400 was announced. The AS/400 was developed to combine the ease-of-use, large application base, and connectivity options of System/36, with the virtual storage management, integrated relational database, and object-based design of System/38. Most application programs developed for the System/36 and System/38 are easily migrated to and run on the AS/400. The vast majority of applications on the AS/400 are written in COBOL or RPG and are designed for transaction processing. Therefore, the AS/400 is optimized for input/output processing rather than for complex calculating.

Version 1 Release 1 of AS/400 was introduced in 1988. The initial hardware models were called the B models. About every 9 to 15 months, a new series of models is announced with the objective of improving price-performance.

In 1991, the Version 2 hardware was announced, which is the platform for the evaluated product. Version 2 Release 3 (V2R3) Feature code 1920 is the software that was initially evaluated. V2R3 was made available to customers in December of 1993 and received the C2 rating in October of 1995.

In October of 1996, the V3R0.5 software running on the Advanced Series Hardware satisfied the Rating Maintenance Phase (RAMP) procedures for a rating of C2. The Advanced Series was a completely repackaged family of hardware that is functionally equivalent to the version 2 hardware. V3R0.5 was made available to customers in June of 1994.

In October of 1997, V3R2 software running on the Advanced Series Hardware satisfied the RAMP procedures for a rating of C2. The hardware portion of the TCB remained the same as the V3R0.5 RAMPed configuration. The software was enhanced to support the porting of POSIX applications, including a hierarchal file system, an execute bit, and multiple group capability. V3R2 was made available to customers in July of 1996.

The current RAMP cycle is to V4R1, which include changes from V3R6 and V3R7, running on the Advanced Series RISC hardware. V4R1 was made available to customers in August of 1997.

The AS/400 family uses a single, integrated operating system called Operating System/400 (OS/400). To meet customer expectations for turnkey operation, OS/400 provides a comprehensive, fully integrated set of batch and interactive processing capabilities to support application program development and execution. OS/400 includes such functions as a relational database, a compilable command language (CL), and security.

2.2. High-Level Overview

The AS/400 software architecture supports system components at three software layers divided by two interface layers. The AS/400 software and firmware is divided into *components*. System documentation covers each component, and software configuration management is at the component level.

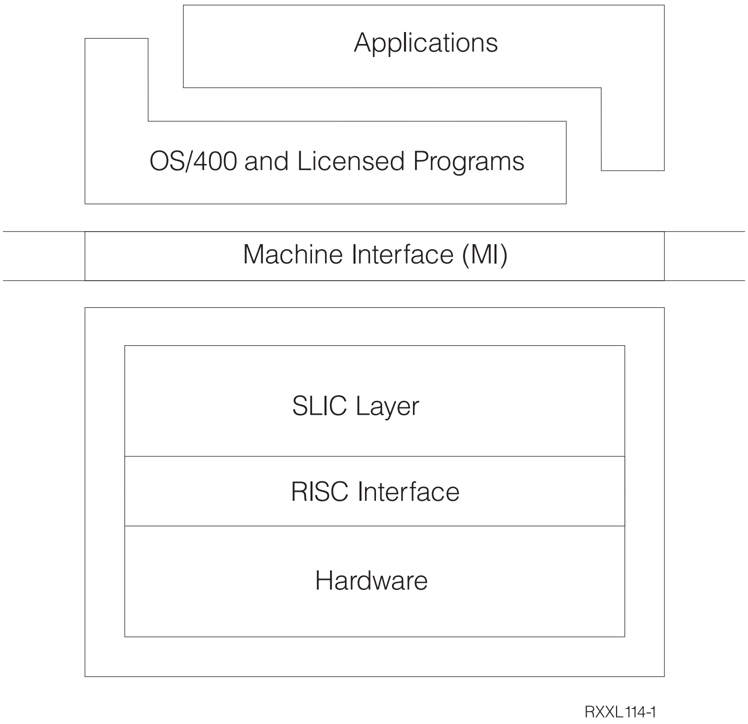


Figure 2-1. AS/400 Overview

Figure 2-1. “AS/400 Overview” on page 2-2 shows the layered architecture of AS/400. The following paragraphs discuss the figure starting from the top and continuing to the bottom. The top layer is Operating System/400 (OS/400) and the languages, utilities, and applications it supports. IBM separates licensed products (LPs) from OS/400; this document uses OS/400 to mean both the operating system and the separately purchased LPs listed in Appendix B, “Evaluated Software”, on page B-1.

From the user’s point of view, OS/400 provides two interfaces for users: menus that allow selections, and direct entry of commands with parameter strings. The menus are based on programs or commands, while the direct entry of commands provides a means of interactively requesting services from OS/400. In addition, a set of programmatic interfaces provides a means for programs to request services from OS/400.

Services provided by OS/400 include storage management functions, job management functions, database management functions, I/O management functions, exception and event handling functions, and message handling functions. Components within OS/400 are responsible for providing these functions by operating in defined ways on a defined set of objects. (These functional areas are also supported by the layers below OS/400.)

To the end user, a unit of work on the AS/400 is known as a job. At the execution phase of a job, OS/400 causes a process to be initiated and the job becomes an *active job*. The active job-to-process relationship is one-to-one. Section 4.1 “OS/400” on page 4-1 covers the interfaces and functions of OS/400.

Unlike other systems in which the operating system interfaces directly with the hardware, AS/400 has a virtual machine interface (MI) under its operating system layer. When more conventional operating systems issue an instruction for execution by the hardware, OS/400 issues an MI instruction. This design makes MI look like the hardware interface to users, and also makes it possible to replace everything under the MI without affecting the user interfaces, user applications, or OS/400 itself.

The MI provides primitive instructions, two program models (original and new), and late binding capability. The MI also provides a consistent interface to low-level services. It is a logical, not a physical, interface, and is not executable. MI instructions are translated to Reduced Instruction Set Computer (RISC) executable instructions. The MI is described more fully in section 3.2 “SLIC and MI” on page 3-16.

Figure 3-8. “The SLIC Layer” on page 3-16 shows the layers below the MI. These are the System Licensed Internal Code (SLIC) layer and the hardware. The RISC instruction set is the interface between the SLIC and hardware layers. When this report refers to “the machine”, everything shown in the figure below the MI is included. MI instructions are converted by the SLIC into RISC instructions.

The SLIC layer is made up of components that operate in defined ways upon a defined set of objects. These components include storage management, process management, task management, I/O management, exception management, interprocess communication, and database management support. MI objects and the MI instructions that operate upon those objects are owned by specific SLIC components. These SLIC components take the MI instructions they are designed to handle from the OS/400 layer and execute them in RISC instructions. The SLIC layer also provides a translator to translate the MI instructions in user-written programs into RISC code.

A process is known to the SLIC as a task. The SLIC layer sees both MI processes and SLIC tasks. Some SLIC tasks do not have a process or job associated with them; an example is an I/O task. A more detailed picture of the SLIC layer is provided by Figure 3-8. “The SLIC Layer” on page 3-16. Section 3.2 “SLIC and MI” on page 3-16 describes the SLIC layer more fully.

The hardware supports a RISC instruction set. At execution time, the RISC instructions are interpreted and executed by the hardware. The RISC interface and the hardware are described more fully in section 3.1 “Hardware” on page 3-1.

2.2.1. Distinguishing Features

An understanding of terms used to describe the AS/400 system will make the sections that follow more understandable. Some terms are familiar but are used with different meanings. These include the following:

- Object, which is used not in the Trusted Computing System Evaluation Criteria (TCSEC) sense but in the sense of object-based
- Active job, which is synonymous with process
- Main storage, which is main memory
- System pointer, which contains a virtual storage pointer and may include authorizations (permissions) as well. System pointers are a type of MI pointer, and are further discussed in section 2.2.6 “Tagged Pointer Architecture” on page 2-6.

Other terms are specific to AS/400. These terms include special authority, private authority, program adoption, exit programs, and blocked instructions. In addition, terms that describe the object-based program template model are also included here.

An *authority* is a type of access permission, such as read or write, explicitly granted to a particular user profile for a unique object.

A *special authority* is a user attribute signifying that the user has special access privileges. For example, the *AUDIT special authority allows the holder to specify system-wide audit criteria, individual object audit criteria, and individual user audit criteria. In addition, the holder of this special authority can retrieve audit records and manage the audit journals.

Program adoption of authority allows a program to run with the authority of the program’s owner and the authority of the signed-on user.

An *exit program* is a user-written program called from a defined point in an OS/400 system program called an exit point. The exit program can be supplied by IBM as a default, or it can be site-defined. For some exit points, an exit program is optional. Exit programs are a way of providing installation unique options to the operating system. The exit program is registered in some way so that OS/400 knows that it exists; it may be a command parameter or in a system value, for example.

A *blocked instruction* is an MI instruction that cannot be translated in the evaluated configuration. The translator included in the evaluated configuration cannot translate the blocked instruction; therefore, it cannot be included in user programs translated on the evaluated configuration. Most blocked instructions are also blocked at runtime in user state (defined below); that is, they cannot be executed in the evaluated configuration except as part of the TCB.

The *Integrated Language Environment (ILE)* supports both *Original Machine Interface (OMI)* and the *New Machine Interface (NMI)*. OMI is the original program template syntax for MI. For OMI, the translator produces encapsulated RISC-executable programs. OMI instructions are storage-to-storage instructions that use a complex instruction set. Calls from OMI programs are dynamically bound at run time. NMI is a new program template syntax for MI. For NMI, the translator produces an MI module object containing RISC instructions that must be bound before it can be executed. NMI instructions are stack-oriented. NMI programs can make program calls that use dynamic run-time binding. NMI programs can also make procedure calls that are bound prior to execution.

Varied on means to make a device available at the user interface of OS/400. This involves actions from the hardware, SLIC, and OS/400 levels.

Varied off means to make the device unavailable. The hardware is deactivated and changes are made to data structures at the SLIC and OS/400 levels.

More information about all of the terms introduced here appears in the sections that follow. The terms state and domain, object-based or object-oriented, and single-level store are discussed in the following paragraphs.

2.2.2. State and Domain

IBM introduced the ideas of *state* and *domain* as an isolation mechanism to protect resources. The program state (which becomes the process and task state) is either system or user. Processes running in system state can execute blocked MI instructions.

The domain is an attribute of the object; objects can be either system or user domain. User state processes cannot directly access system domain objects, but system state processes can access both system and user domain objects. See the table below.

Process State	Object Domain	Object Access
User	User	Allowed (subject to DAC checks)
User	System	Not Allowed
System	User	Allowed (subject to DAC checks)
System	System	Allowed (subject to DAC checks)

2.2.3. Object-Based

Traditionally, systems have been seen in process-oriented terms. A system is seen as a collection of processes, each responsible for a specific portion of the overall task. The processes operate on data flowing through the system. In contrast, the building blocks of a system in object-oriented terms are objects and messages. A system is seen through its structural aspects. The system’s functional aspects are represented by computations involving objects and communications between objects.

AS/400 is characterized as “object-based” rather than “object-oriented.” The term “object-oriented” implies certain concepts, as follows:

- Objects belong to a type or class hierarchy. A class represents all objects with some set of similar characteristics. A subclass represents all objects that belong to the same superclass and that possess some additional common characteristics. An object inherits characteristics from its superclass.
- Objects are encapsulated; that is, operations are defined for object classes at the interface to that object class. Other operations, since they are not defined for the object class, are not performed or even understood at the interface.
- Objects communicate by sending and receiving messages. Sending a message to an object with the name of a specific operation available at the object class’s interface is the only mechanism by which an object can manipulate another object’s encapsulated data. The object-oriented model provides data and program abstraction, hiding details of an object’s implementation from other objects.

The term “object-based” is more appropriate for AS/400 than the term object-oriented because AS/400 does not support class hierarchies or inheritance. Instead, AS/400 provides strong object typing. From the user’s point of view, all data structures, devices, and other resources are objects. Objects are instantiated from predefined templates. The operations provided by the object manager for each type of object determine what a user can do to and with that object. Objects in AS/400 are encapsulated, so data and program abstractions are provided by AS/400.

The AS/400 MI instruction set provides an object-based interface. When a user program requests an object that has been referred to by name, a pointer to the object is *resolved*. A *materialize* operation can then make the encapsulated object's attributes available to the user program.

2.2.4. Integrated Database Management System

The AS/400 Database Management System (DBMS) is fully integrated with the AS/400 software at each layer. There is no single definable piece called a DBMS. Instead, there are components at the OS/400 and SLIC layers that provide database management system functions. For example, at the OS/400 layer, there are direct commands to create, destroy, and display database files, as well as a query interface and a Structured Query Language (SQL) interface, among others. At the SLIC layer there are components that support these interfaces and database functions such as journal management. In addition, the DBMS uses the same SLIC components to provide I/O and other services that are used by all components.

2.2.5. Single-Level Store

Single-level store means that the AS/400 has a single system-wide virtual address space. From the user's point of view, storage is not divided into main storage and disk storage. Instead, the virtual storage is large enough to contain all data to be stored on the system.

Functions operating at the level that the user sees (that is, at the OS/400 level) have a view of objects as stored in contiguously addressable locations within the single-level store. Space for an object is allocated as connected virtual storage. In reality, the data representing the object resides on auxiliary storage in disk extents that are not necessarily contiguous. Below the MI, the Storage Management component of SLIC translates the virtual storage address to a request to the I/O management SLIC components. This request is passed to the hardware. When the I/O request is fulfilled, the object is materialized and it becomes available above the MI.

The Storage Management component also manages main storage. The SLIC level and lower levels are below the reach of all but IBM system programmers and cannot be affected by any user of the evaluated configuration, whether trusted or not. Single-level store, auxiliary storage management, and main storage management are covered in greater detail in section 3.2.3.3 "Storage Management" on page 3-47. Resolving pointers and object management are covered in section 3.2.1 "Object Management in SLIC" on page 3-17.

2.2.6. Tagged Pointer Architecture

In a system using single-level store, a pointer is extremely powerful because it can be used to access any virtual storage location. In addition, an AS/400 pointer contains a virtual storage pointer and may include authorizations (permissions). AS/400 provides hardware protection for all pointers addressable by an MI instruction operand.

Associated with every 8 bytes of main storage is an additional tag bit that is not addressable above the MI. These tag bits are tested by a specific group of RISC instructions when a pointer value is loaded from the associated 16 bytes of main storage to see if it is a valid pointer value.

Any attempt to modify a tagged pointer, except with the proper MI instructions, causes the bit to be cleared and the pointer to become invalid. There is no interface to repair the tag bit; instead, a new pointer must be created by using the proper MI instruction. This prevents non-pointer data from being used as pointer data. See section 3.1.1.1.2 "Tag Bits" on page 3-8 and section 3.2.1.2.4 "MI Pointers" on page 3-21 for more information about tagged pointers.

2.2.7. Hardware Storage Protection

The Storage Management component of SLIC sets bits reserved for storage protection when it places data in main storage. These bits are associated with frames and protect data in main storage as no read/write from user state, read-only from user state, read-only from any state, or read/write allowed from any state. (See section 3.2.3.3 “Storage Management” on page 3-47 for a discussion of frames.) When an address is translated, the setting of the storage protection bits is examined. For more about hardware storage protection, see section 3.1.1.1.3 “Hardware Storage Protection” on page 3-10.

2.3. Trusted Computing Base (TCB)

2.3.1. TCB Structure and Definition

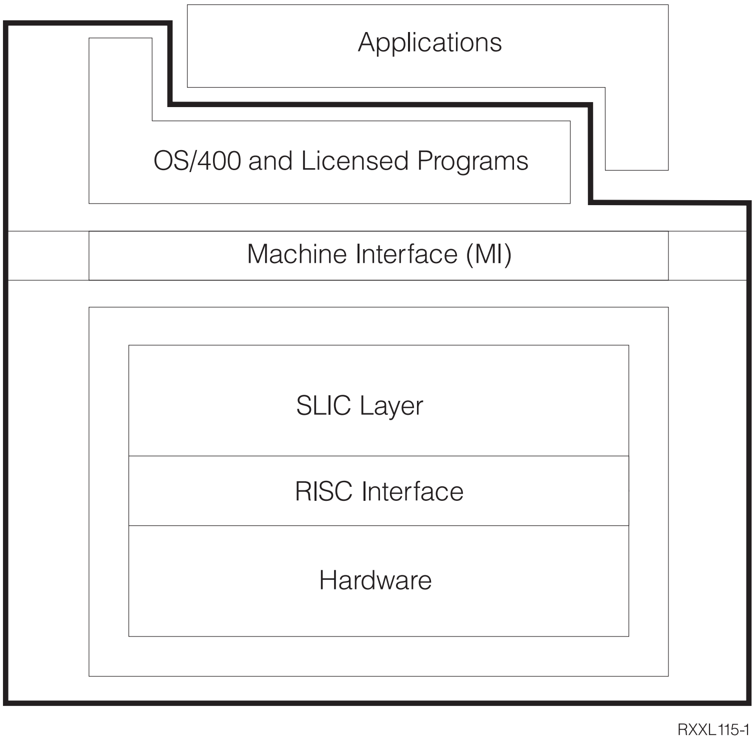


Figure 2-2. TCB Boundary

The TCB consists of all programs that run in system state and trusted processes. Figure 2-2. “TCB Boundary” on page 2-7 shows the TCB boundary. Within this boundary are the OS/400, the MI, the System Licensed Internal Code (SLIC) layer, the RISC interface, and the hardware. There are three interfaces to the TCB: the Control Language (CL), the Application Programming Interface (APIs), and the Machine Interface (MI) instructions. (The box shown in the figure as “Applications” is intended to represent those programs that run in user state. The extension to the right of the box represents those user state programs that issue MI instructions without going through OS/400.)

2.3.1.1. TCB Identification

The TCB is comprised of the components that enforce the security policy and that can possibly violate the security policy. The AS/400 TCB consists of the AS/400 hardware platform and portions of the AS/400 software.

Appendix A, "Evaluated Hardware", on page A-1 contains a complete list of hardware in the evaluated configuration, and the AS/400 hardware is described in section 3.1 "Hardware" on page 3-1. The hardware is in the TCB because it is trusted to operate correctly.

The AS/400 software consists of SLIC software and OS/400 software, as described in section 3.2 "SLIC and MI" on page 3-16 and section 4.1 "OS/400" on page 4-1, respectively. Both SLIC and OS/400 are comprised of *components* with each component providing certain functionality. For example, the SLIC Storage Management (SM) component provides Storage Management functionality, and the OS/400 Spool component provides print spooling functionality.

When the AS/400 system is configured for C2, the code for some components, although installed, is not executed. At the OS/400 layer, the code in these components is not executed because the interfaces to these components (either CL commands or APIs) are excluded from use by DAC (PUBLIC authority is *EXCLUDE), and administrators are warned in the TFM not to use these interfaces and not to change the authorities on these interfaces.

At the SLIC layer, the code in certain SLIC component(s) is not executed because the component provides no unblocked MI instructions. Therefore, the only interfaces to these components are blocked MI instructions, and the only OS/400 components that use these blocked MI instructions are the excluded OS/400 components described in the previous paragraph.

The SLIC and OS/400 components containing code that can be executed when the AS/400 system is configured for C2 are identified in Appendix B, "Evaluated Software", on page B-1.

When the AS/400 System is configured for C2, all SLIC components that can be executed are part of the AS/400 TCB. SLIC, which implements the Machine Interface (MI), is described in section 3.2 "SLIC and MI" on page 3-16. The SLIC software, which is generally written in either an IBM internal language (which is a derivative of PL/1) or C++, is developed by IBM using special-purpose compilers to produce the RISC instructions. The SLIC modules perform such functions as storage management, process management, and object management. In addition, SLIC enforces DAC, as described in section 6.2 "Discretionary Access Control" on page 6-4, and supports audit functionality, as described in section 6.5 "Audit" on page 6-18. SLIC also manages the object attributes of hardware protection class and domain. Finally (since the AS/400 hardware does not support hardware states), SLIC manages process states (see section 4.1.2.2 "Programmatic Interfaces" on page 4-15). SLIC consists of approximately 2.8 million lines of executable code.

OS/400, which is generally written in an IBM internal language, which is a derivative of PL/1, consists of approximately 11.5 million lines of executable code. Each OS/400 component is made up of some combination of UDSS, UDIS, SDSS, and UDUS *PGM objects. All UDSS and SDSS programs contained in these components are also part of the AS/400 TCB. In addition, all UDIS programs are part of the TCB when invoked from System State. Finally, UDIS and UDUS programs that run with any of the special authorities described in section 5.2.6 "User Classes and Special Authorities" on page 5-4 are also part of the AS/400 TCB. The rationale for these statements is described below:

- All System Domain/System State (SDSS) program objects are part of the TCB. These programs implement CL commands and perform other functions such as spooling support and Identification and Authentication (IA). SDSS programs are in the TCB because, as discussed previously, System State programs cause the invoking process to execute in System State. When a process is executing in System State, it can execute blocked MI instructions, and it can read and write main storage that is hardware storage protected read only or no read/write. SDSS programs are described in section 4.1.2.2.3 “System Domain System State Programs” on page 4-16.

- All User Domain System State (UDSS) program objects are part of the TCB. UDSS programs are in the TCB because they cause a process to execute in System State. In addition (since they can be called by non-TCB programs), they are “trusted” to protect the TCB by performing parameter validation as described in section 4.1.2.2.4 “User Domain System State Programs” on page 4-16. UDSS programs are described in section 4.1.2.2.4 “User Domain System State Programs” on page 4-16.
- User Domain Inherit State (UDIS) program objects are part of the TCB when they are invoked from System State because they execute in System State. When they are invoked from User State, they are not part of the TCB because they run in User State and; therefore, cannot violate the security policy. UDIS program objects are described in section 4.1.2.2.1 “User Domain Inherit State Programs” on page 4-16.
- All UDIS program objects that adopt an IBM-supplied user profile are part of the TCB. These programs are part of the TCB even when invoked from User State because they run with:
  - any special authorities associated with the IBM-supplied user profile
  - owner access to any objects owned by the IBM-supplied user profile
  - access to any objects to which the IBM-supplied user profile has authority.
- All User Domain User State (UDUS) program objects that adopt an IBM-supplied user profile are part of the TCB. These programs are part of the TCB even though their invoking process runs in User State because they run with:
  - any special authorities associated with the IBM-supplied user profile
  - owner access to any objects owned by the IBM-supplied user profile
  - access to any objects to which the IBM-supplied user profile has authority.

  UDUS programs objects are described in section 4.1.2.2.2 “User Domain User State Programs” on page 4-16.

In addition, *trusted processes* are included as part of the TCB. At the OS/400 level, there is a set of trusted processes, known as system jobs, that are also part of the AS/400 TCB. The trusted processes include processes that are started as part of system initialization (e.g., QSPLMAINT) and others that are started as a result of a CL command (e.g., spooling writers) or an API. These processes are included as part of the TCB because they meet at least one of the following conditions:

- The process enforces the security policy.
- The process runs with a user profile that has special authorities that allow the process to run unconstrained with respect to part of the security policy.

In addition to the executable code that is part of the TCB, some objects are part of the TCB. Any objects containing data used by the TCB to enforce the security policy are part of the TCB. For example, user profile objects are part of the TCB because they contain security relevant information about users on the system and because the TCB “trusts” that this information is correct.

Finally, all object types have security-relevant information in their segment headers and EPA that is trusted by the TCB to be correct and used by the TCB in making security relevant decisions.

2.3.2. TCB Interfaces

This section describes how users or subjects (i.e., jobs acting on behalf of users) request services from various parts of the TCB. There are four interfaces through which TCB services can be requested. One of these interfaces provides an interactive user interface. The other three interfaces are programmatic interfaces, which provide mechanisms for a UDUS program executing within a process to request TCB services. Each interface is described below.

CL Command - CL commands are described in section 4.1.2.1 “Command Language” on page 4-14. As discussed in this section, IBM supplies a set of CL commands and provides a mechanism by which users can create additional CL commands. The CL commands developed by users can only invoke programs that run in US. Only IBM-supplied CL commands are TCB interfaces.

UDSS program - When the UDSS programs begin execution, the state of the process changes from User State to System State. UDSS programs are referred to by IBM as APIs. They are described in more detail in section 4.1.2.2.4 “User Domain System State Programs” on page 4-16. A SYSCALL mechanism is also used to transition states. Details can be found in section 4.1.2.2.5 “Kernel Mode” on page 4-17.

UDUS program or UDIS program that adopts an IBM-supplied user profile - While this program is running, the user’s process has any special authorities associated with the IBM-supplied user profile, and it has access to any objects owned by the IBM-supplied user profile or to which the IBM-supplied user profile has authority.

Unblocked MI instruction - Unblocked MI instructions may either map directly to RISC instructions or they may be calls to SLIC routines. Consequently, unblocked MI instructions provide a direct interface to SLIC. Unblocked MI instructions are described in section 3.2.2.2.3 “Unblocked Instructions” on page 3-40.

2.3.3. TCB Policy

AS/400 is designed to provide C2-level security in a multiuser environment. This section provides a brief summary of how the C2 requirements are met.

Discretionary access control (DAC) restricts access to objects based on the identity of a user and a user’s group. AS/400 does this by object authority, which is designed to prevent unauthorized access to objects. Special authority allows trusted users to have access to objects. More about DAC is found in section 6.2 “Discretionary Access Control” on page 6-4.

Object reuse prevents a user who creates a new object on the system from accessing any information that formerly resided in the same location in auxiliary storage or main storage. This objective is accomplished by having storage cleared on allocation before reuse. In addition, the single-level store and management of virtual addresses ensure that two objects never share the same virtual address space. Memory pages are cleared before a new page is moved in. More about object reuse is found in section 6.4 “Object Reuse” on page 6-17.

Identification and Authentication require that system users be uniquely and unambiguously identified. On the AS/400 system, each user is uniquely identified by a user profile and authenticated by a password. Batch jobs are also associated with a specific user profile. More about Identification and Authentication is found in section 6.1 “Identification and Authentication” on page 6-1.

The AS/400 audit mechanism provides a way to audit security-relevant events performed by system users. A combination of system values, user profile parameters, and object parameters is used to determine which events are audited. More about audit is found in section 6.5 “Audit” on page 6-18.

System Architecture was identified by the Preliminary Technical Review as a point of concern because the AS/400 system runs on single-state hardware with no support for separate user and kernel modes. Instead, the AS/400 uses the software implementation of state and domain, introduced above, to support two execution states. In addition, the hardware storage protection protects against unauthorized changes to TCB code. For more about how the system architecture requirement is satisfied, see section 7.1 “System Architecture” on page 7-1.

Section 3
Machine Architecture

3.1. Hardware

This section describes the hardware platform, including the supporting microcode that is included in the AS/400 Trusted Computing Base (TCB). This hardware and microcode is designed and implemented by IBM for the express purpose of supporting the AS/400 system. As described in Appendix A, "Evaluated Hardware", on page A-1, AS/400 comes in a variety of series, machines types, and models.

3.1.1. Hardware Configuration

Functionally, to the user, all AS/400 models are exactly the same. The only differences are performance and the number of attachable peripherals. They all contain the same hardware features, such as hardware storage protection, but differ internally regarding the use of cache, execution units, and number of processors. All models run the same System Licensed Internal Code (SLIC), run the same OS/400 operating system, and run with the same input/output processor (IOP) microcode loads.

Each AS/400 evaluated model contains one or more of the following: RISC processor, cache, System Products Division (SPD) bus, main storage, input/output processors, service processor, and peripheral devices. These are shown in Figure 3-1. "Hardware Layer" on page 3-2. Each of these is discussed in detail later in this section. A functional diagram showing the interrelationships of the hardware units that make up the basic system configuration models[1] is shown in Figure 3-2. "Basic System Configuration Hardware" on page 3-3.

At the high end of the models, multiple processors may be present, and additional hardware is present to increase performance. The hardware that makes up a complex system configuration model is shown in Figure 3-3. "Complex System Configuration Hardware" on page 3-4.

3.1.1.1. The AS/400 CPU

The Central Processing Unit (CPU) hardware consists of the various processor execution units (Load/Store, Branch, Arithmetic, Floating Point and other miscellaneous execution units), cache, and a storage control unit (memory control). These are on or closely attached to the processor card, as shown in Figure 3-2. "Basic System Configuration Hardware" on page 3-3 and Figure 3-3. "Complex System Configuration Hardware" on page 3-4.

[1] In the initial models and version of the AS/400 hardware, "Low-End Processor" and "High-End Processor" were used to describe specific technology, processor, and packaging configurations. As the product has matured, this is no longer true. With today's architecture, there are Basic System Configurations, similar to the "Low-End Processor", in each of the machine types (9401, 9402, 9406) and packaging options. The Complex System Configurations, similar to the "High-End Processor", can be found in 9406 model types with the larger packaging options.

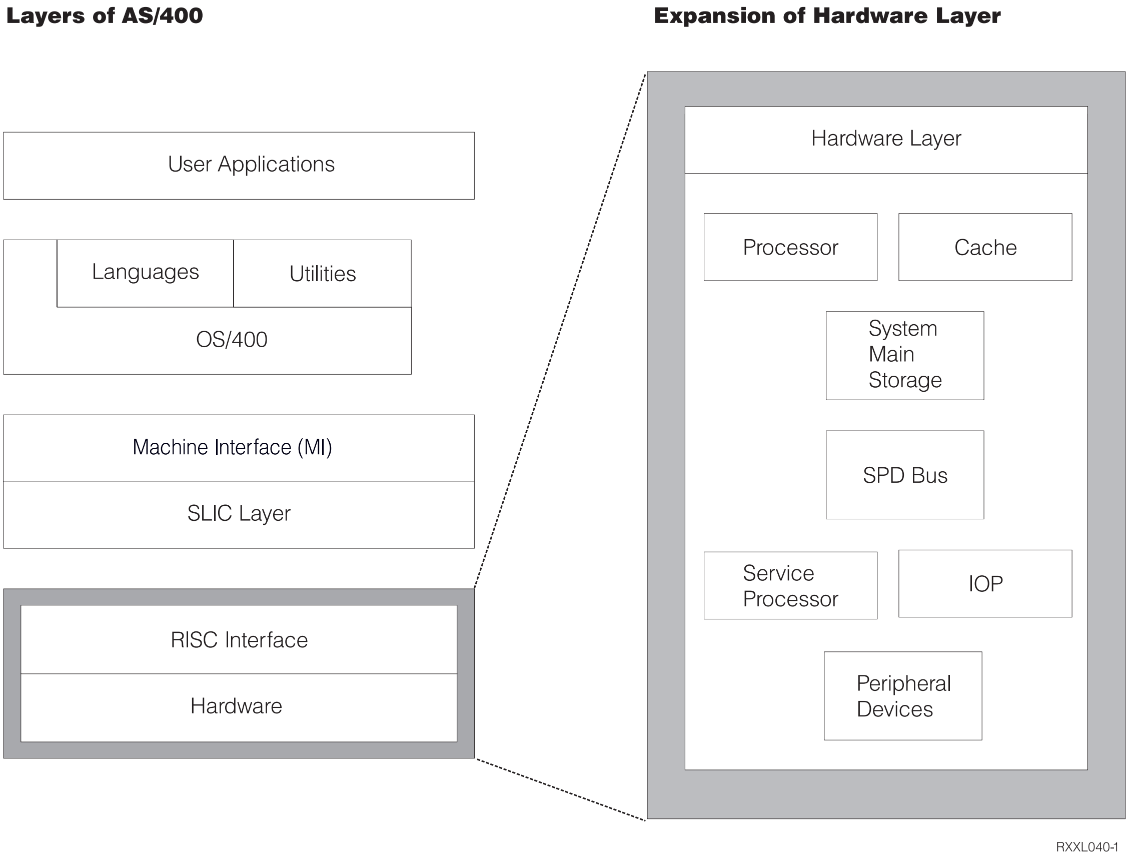


Figure 3-1. Hardware Layer

The AS/400 CPU hardware directly executes the Amazon Architecture Instruction set (a superset[2] of the PowerPC Instruction set). The Amazon Architecture is a 64-bit Reduced Instruction Set Computer (RISC) architecture. Since the AS/400 user codes to an abstract interface, the user has no explicit interface available to the hardware.

The processor organization with component relationships is shown in Figure 3-4. “AS/400 Processor” on page 3-5 and is described in the following paragraphs.

The Amazon processors are superscalar register-to-register architecture with 1 to 4 instructions that can be dispatched each cycle to the 5 instruction execution units, depending on the order in which the instructions appear in the program. The instructions are executed to completion through multiple staged pipes with wait states used to hold execution in a stage when data is not available to the pipe stage. All instructions are executed in order and cache misses cause instructions following a Load/Store operation to be stalled. All Amazon instructions are 32 bits in length and all general purpose registers are 64 bits in length.

[2] The Amazon Architecture has an additional 22 instructions to support tag bits, decimal operations, more efficient data movement, and system call return. In addition, the Amazon Architecture uses a different virtual address translation method to support the AS/400 single level store.

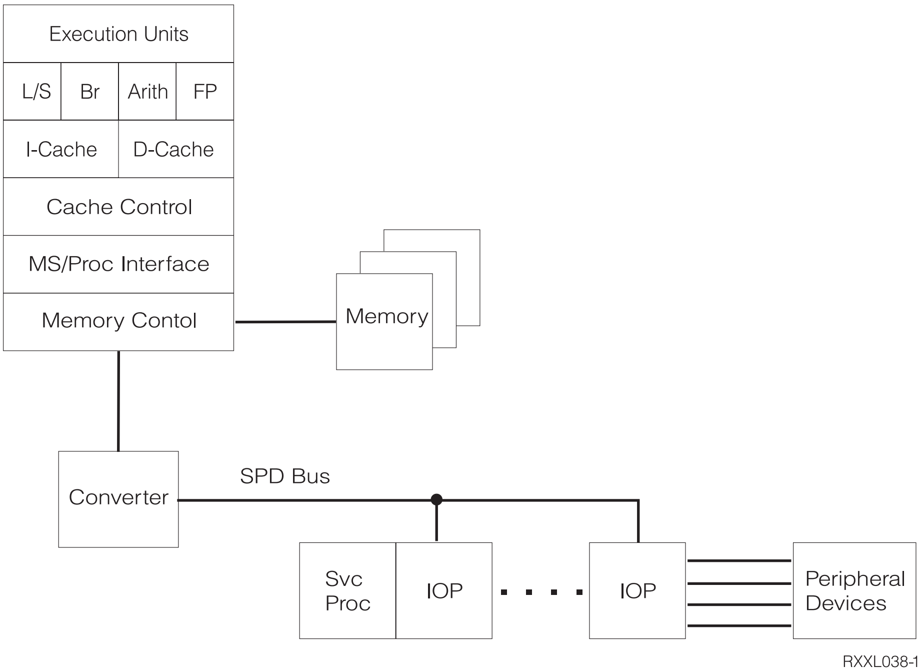


Figure 3-2. Basic System Configuration Hardware

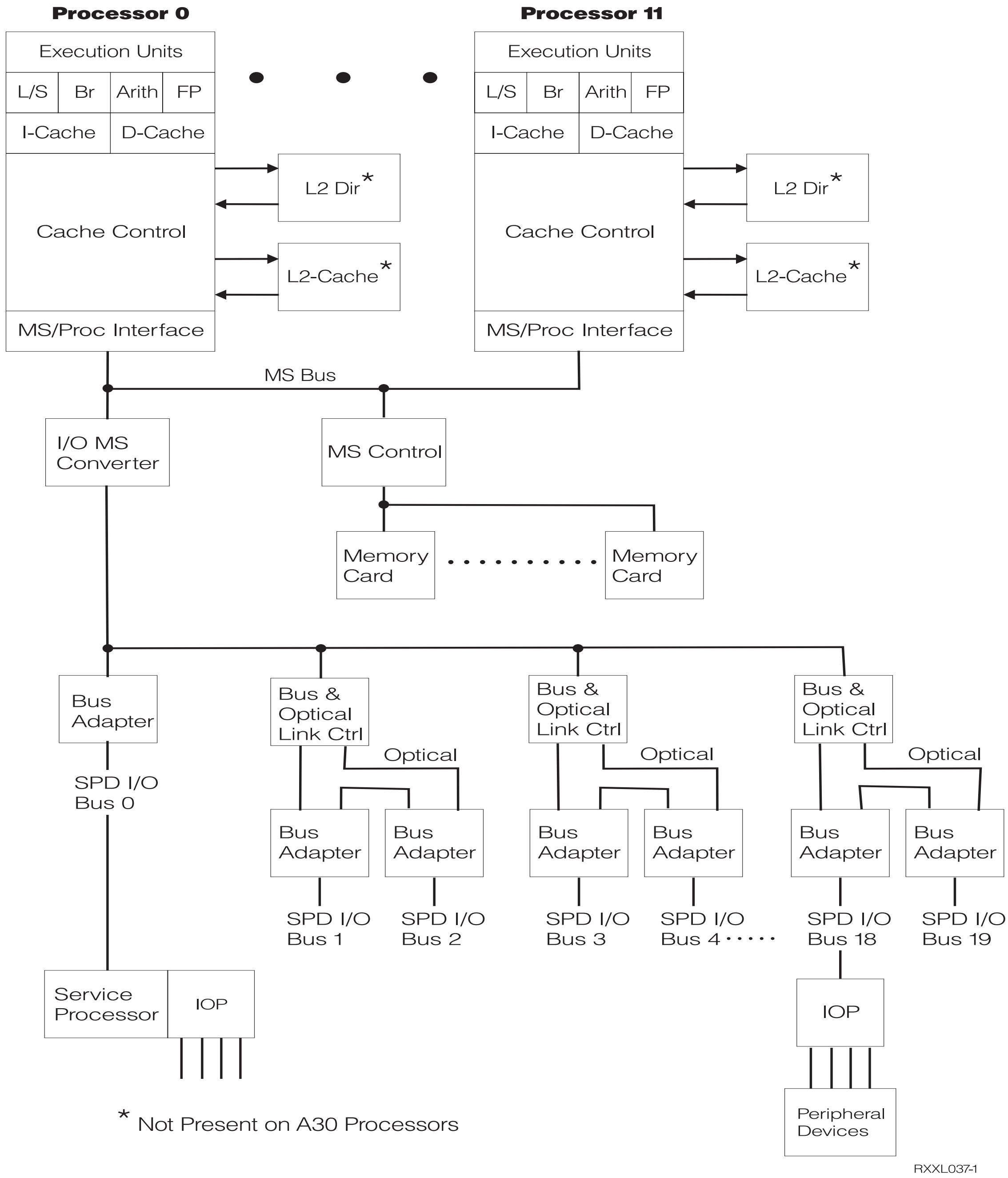


Figure 3-3. Complex System Configuration Hardware

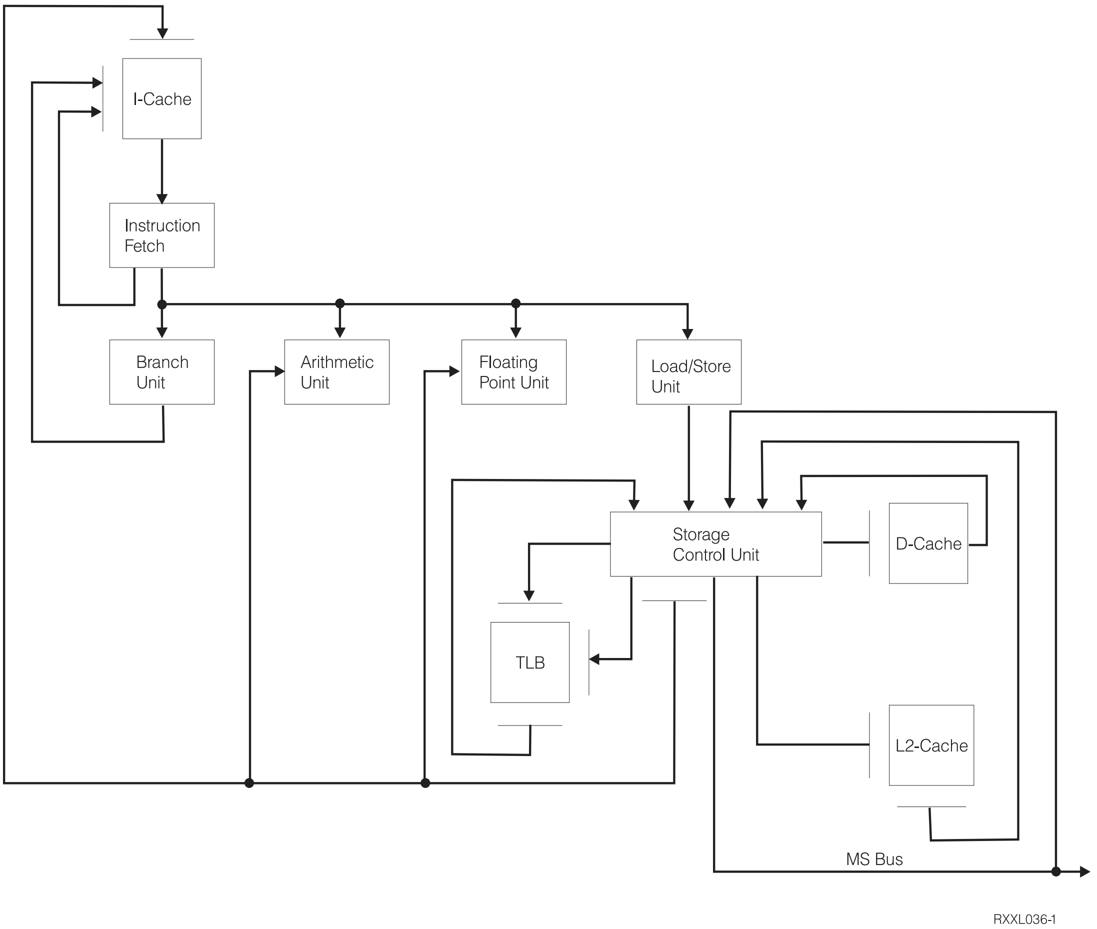


Figure 3-4. AS/400 Processor

The processing unit is made up of several components, including the following:

- Instruction Fetch Unit, which fetches instructions from the Instruction cache and places them into an 8 to 16 entry Instruction Buffer. Each instruction from the Instruction Buffer is routed to the appropriate execution unit to execute the instruction.
- Branch Unit, which searches the incoming instruction stream for branches and attempts to fetch instructions for the branch direction most likely to be taken. The A30 processor design does not use Branch History Tables to help define the branch leg most likely taken.
- Load/Store unit, which handles fixed point and floating point loads and stores and handles the cache management functions.
- Arithmetic unit which handles most of the fixed point arithmetic instructions.
- Floating Point unit, which handles the IEEE 64-bit (double precision) format. All IEEE 32-bit formats are converted to IEEE 64-bit format as they are loaded or stored to or from the Floating Point unit by single precision floating point instructions. The floating point unit supports the IEEE 754 floating point standard.
- Address Translation Unit, which translates all addresses from the Load/Store Unit and the Instruction Fetch unit into real addresses used to access the L1 instruction cache, L1 data cache, L2 cache and main storage. (L1 and L2 denotes industry standard notation where L1 is a smaller cache but faster than L2 cache.)
- Cache Control units, which manage the data coherency in the various Data caches as well as fetching requested cache lines into the caches and storing or invalidating various cache lines that need to be moved to higher level caches or to memory. The cache control units handle all coherency items relating to the data in the caches. Coherency is NOT maintained in the I-caches by the hardware. SLIC must manage the coherency by invalidating the I-cache contents if instructions are modified.
- L1 I-Cache, which holds recently referenced instructions, has several sizes from 4 KB to 64 KB, with parity detection. If a parity error is detected, the cache line is invalidated and the instruction stream is refetched from memory, with the required instruction bypassing the cache and sent directly to the Instruction decode unit (thus allowing the system to continue to run even with a solid I-cache error). The cache directory is checked during fetch operations; if the instruction is in I-cache, it is loaded; otherwise, a cache miss occurs. When a cache miss occurs, the storage control unit fetches a 64 or 128 byte block[3] of instructions from main storage and stores the data in the I-cache. The AS/400 I-cache cannot be written other than by modifying the instruction stream in the D-cache, storing the D-cache entry back to main storage, and then purging the I-cache entry so the instruction is refetched. The I-cache is addressed via real addresses.
- L1 D-Cache, which holds recently referenced main storage data, has several sizes of 4 KB, 8 KB, 64 KB, and 256 KB and has Single Bit Error Correct (SBEC) Error Correcting Code (ECC). The D-cache directory is checked during fetch operations; if the data is in the D-cache, the data is loaded in the register; otherwise, a cache miss occurs. When a cache miss occurs, the storage control unit fetches a 32 byte, 64 byte or 128 byte block[4] of data from L2, if available, or main storage when it is not available in L2, and stores the data in the D-cache. The AS/400 cache is implemented as a store-in cache, which means that all write operations are initially stored in the D-cache and then copied to main storage when the cache line is purged either at the request of another processor or to free up a cache line for use by another data request. All processors that support multiple processors, place all real main store write requests that are not modified or exclusive on a common bus shared

[3] The size of the instruction block fetched is fixed (always 64 or always 128) by machine model based on the cache line size.

[4] The size of the data block fetched is fixed (always 32, always 64, or always 128) by machine model.

between processors. The processors can then snoop the address to make sure that another processor does not have a changed copy of the data. The D-cache follows the standard Modified, Exclusive, Share, Invalid (MESI) cache protocol. Write operations, in multiprocessor configurations, result in other processors invalidating, if no change was made to the cache, or copying back to memory a changed cache line corresponding to the main storage location being accessed.

- L2-cache, which holds recently referenced main storage data, has several sizes, from 0 bytes to 4 MB and has single bit error correct (SBEC) ECC. When a cache miss occurs, the storage control unit fetches a 128 byte data block[5] from main storage and stores the data into the L2-cache. Data and instruction fetches are first placed into the L2-cache (if present) when the processor does not have the data in the L1-caches. The processor then transfers the data from the L2-cache to the L1-cache (the I-cache or the D-cache). The L2-cache uses a MESI cache coherency protocol. The L2-cache directory contains the address of all cache lines in the I- and D-caches on the current processors. All processors that support multiple processors place all real main store write requests that are not modified or exclusive on a common bus shared between processors. The processors can then snoop the address to make sure that another processor does not have a changed, exclusive or shared copy of the data. Write operations, in multiprocessor configurations, result in other processors invalidating the L2-cache line (or copying the L2-cache line back to main storage) corresponding to the main storage location being accessed, which in turn invalidates the I- or D-cache line.
- Storage control unit (SCU), which translates virtual addresses used by the processing unit and accessing main storage to fetch or store data. The storage control unit receives effective addresses from the Load/Store unit and translates these addresses to real main store addresses. The SCU then searches the D-cache for the associated address or, if not found in the D-cache, the SCU searches the L2-cache for the address. If found in the L2-cache, the SCU moves the data to the D-cache. If the address is not found in the L2-cache, then the SCU initiates a main store (MS) access to fetch the cache line from memory (future designs may also fetch directly from another processor's cache) into the L2- and D-caches. The SCU also generates address snoop requests from and to other processors for all cache accesses that require coherency.

AS/400 CPU Specifics

The evaluated hardware includes three different CPUs: the A10, A30, and A35. The CPUs are all implementations of the Amazon architecture. The A10 and A30 are single chip CMOS processors while the A35 is implemented in a 5 chip BI-CMOS process. The A10, A30, and A35 contain five execution units. They all have a Load/Store unit, a Floating Point unit, a Branch unit, a Fixed Point unit and either a Condition Register unit or a Shift Rotate unit.

The A10 processor memory structure consists of a 4K L1 instruction cache, 4K or 8K L1 data cache, 0 or 1 megabytes of L2 cache, and supports up to 1 gigabyte of memory. The A10 can be configured only as a single processor running up to 90 MHz. The A10 can execute 3 instructions per cycle and can support up to 7 SPD buses.

The A30 processor memory structure consists of a 8K L1 instruction cache, 256K L1 data cache, no L2 cache, and supports up to 4 gigabytes of memory. The A30 can be configured up to a 4-way processor running up to 182 MHz. The A30 can execute 4 instructions per cycle and can support up to 19 SPD buses.

The A35 processor memory structure consists of a 64K L1 instruction cache, 64K L1 data cache, 0 to 4 megabytes of L2 cache, and supports up to 32 gigabytes of memory. The A35 can be configured up to a 12-way processor running up to 125 MHz. The A35 can execute 3 instructions per cycle and can support up to 19 SPD buses.

[5] The size of the data blocked fetched for the L2 cache is always 128 bytes for all machine models.

3.1.1.1.1. Hardware Address Translation

The Amazon architecture supports an 80-bit virtual address space, but the various processors implement a minimum of 64 bits and a varying number of additional virtual address bits, depending on the processor. Figure 3-5. “Hardware Address Translation” on page 3-9 illustrates the hardware address translation. The AS/400 software supports only a 64-bit virtual address space, and the upper 16 bits (bits 0-15 of an 80-bit virtual address) are always 0. All processor General Purpose Registers used for addressing contain 64 bits. These 64 bits can be used to create three different forms of address, E=DS, E=R, and virtual addresses.

- Effective = Direct Store (E=DS)

  This address form is used only in privileged mode. E=DS addresses are defined by the high order 28 bits having a value of X’0000801’ (note that the high order 16 bits of the 80 bit virtual address are always zeros). These addresses are used to address I/O and key registers that are not part of the processor architecture but are used to send and get information from design specific hardware. E=DS addresses are not translated by the processor hardware but are sent to the chips on the MS bus (unless the address is an internal processor register).

- Effective = Real (E=R)

  This address form can be used only in privileged mode. E=R addresses are defined by the high order 28 bits having a value of X’0000800’ and are used to directly address memory with no hardware address translation (maximum of 52 bit real address space).

- Virtual

  The virtual address is formed by concatenating the 64-bit address (in the low-order bit positions) with 16 bits of zeros (in the high-order bit positions) forming the 80 bit address. The virtual address is translated by first checking the Translation Lookaside Buffer (TLB) to determine if the virtual address is located there. The TLB contains the most recently used virtual addresses that were translated. The TLB contains the virtual address and the associated real address. If the virtual address is not in the TLB, then the hardware will go to the Page Table. The location of the Page Table Entry is found by hashing the virtual address and using the hash as an index from the starting Page Table address found in Storage Description Register 1 (SDR1) to refer to the Primary Page Table Group consisting of 8 Page Table Entries. The Page Table Entry contains the virtual address and the real address associated with the virtual address as well as page status information. If the entry is not found in the Primary Page Table Group, then a secondary hash is used to refer to a secondary Page Table Group consisting of 8 more entries of the same format as the Primary Page Table Group. If the virtual address is not found in the Page Table, then a Data Storage Interrupt (or Instruction Storage Interrupt) is generated and SLIC must locate the virtual address from the SLIC maintained page tables and move it into one of the Page Table Entries and then return to the interrupted routine to re-execute the instruction. The hardware copies the real address and virtual address as well as the Page Table Status from the Page Table and put that into the TLB.

Once the real address is known, the Storage Control Unit locates the cache entry addressed by the real address if it is found in the cache directory or causes the referenced cache line to be brought into the cache from memory.

The architecture has special Instruction Relocate (IR) and Data Relocate (DR) bits that, when reset, cause any format of the 64-bit address to be treated as a real address. These bits are reset by the hardware when an interrupt occurs.

3.1.1.1.2. Tag Bits

The hardware chips used in the AS/400 have tag bits.[6] For each 8 bytes of memory, a separate bit called a tag bit is maintained in memory (and cache). These tag bits are used to identify Machine Interface (MI) pointers, which are address pointers that OS/400 and users use to access data and MI objects. (See section 3.2.1.2.4

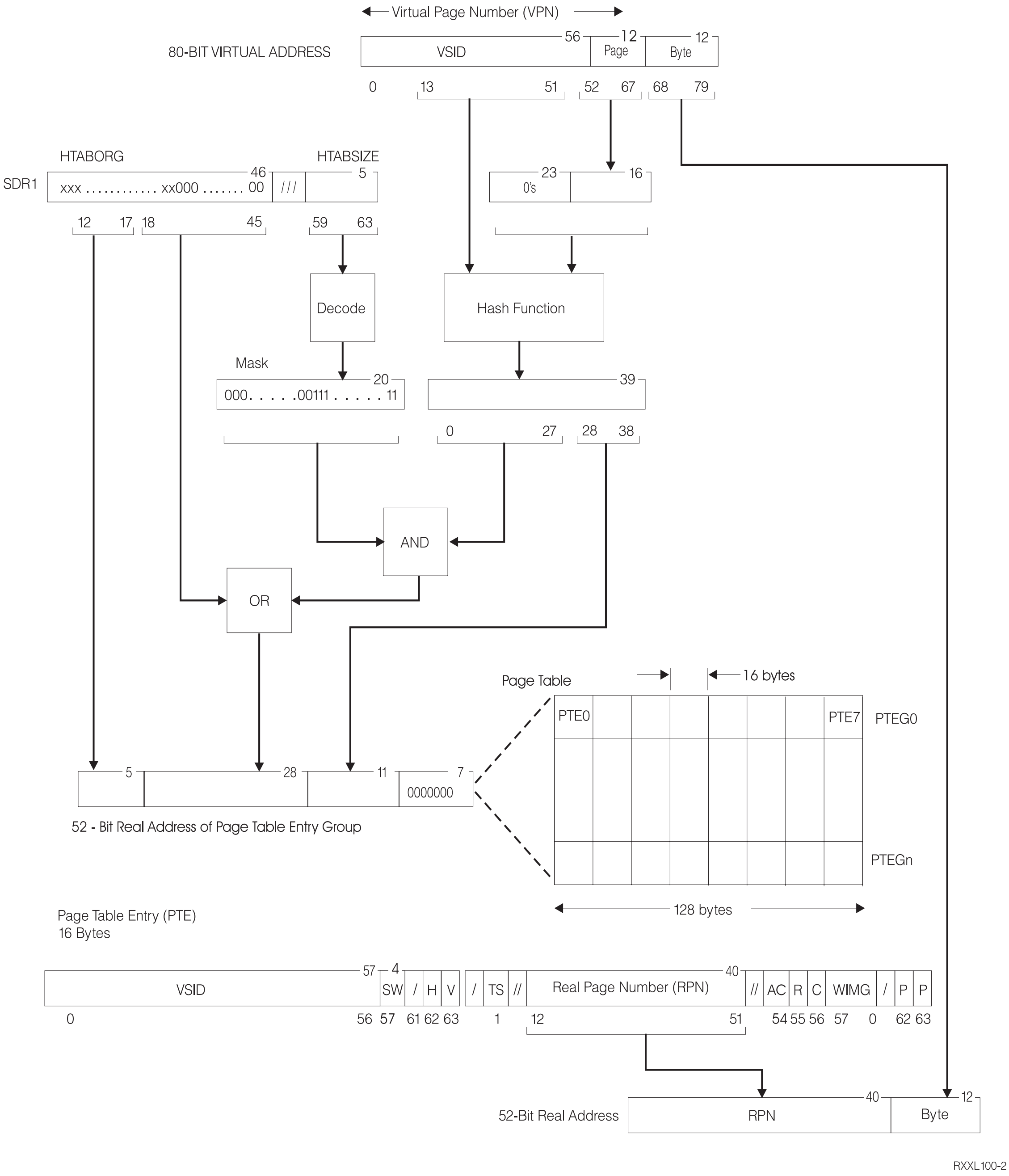


Figure 3-5. Hardware Address Translation

6. The processor is set into tags-active mode by the service processor during IPL as part of the hardware configuration.

"MI Pointers" on page 3-21.) The MI pointer is defined as being 16 bytes in length and 16-byte aligned. To be a valid MI pointer, the tag bits of the two double words must both be set.

When a fetch from memory is performed by a processor, the tag bits are read from memory (or cache) along with the required data. The tag bits for the two double words are ANDed to form a single tag bit, which is placed in the Fixed-Point Exception Register (XER). Table 3-1, "Tag Bits in Memory and XER," on page 10 shows the possible settings for the tag bits in memory (or cache), the XER tag bit resulting from the AND of the memory tag bits, and whether the tag bits indicate that a valid MI pointer (also referred to as a tagged pointer) is present in the two double words (quadword).

Table 3-1. Tag Bits in Memory and XER

Memory or Cache		XER	Valid MI Pointer
Double Word 1 Tag Bit	Double Word 2 Tag Bit		
0	0	0	No
0	1	0	No
1	0	0	No
1	1	1	Yes

When referencing MI pointers, SLIC uses RISC instructions Load Quadword (lq) and Store Quadword (stq) to read the tag bits into the XER or to store the tag bit contained in the XER register into tag bits for each double word. When a store to the memory or cache is done, the tag bits are reset unless the RISC instruction stq is used. They are reset for all other RISC store instructions.

When direct memory access (DMA) saves memory implicitly, the tag bits are not sent to the I/O device. When DMA writes to memory, the tag bits are cleared. To identify when a page has tags, the Page Table Entry has a Tag Set bit (DW1 bit 1) indicating that a tag was set at some time somewhere in the 4096 bytes. (SLIC must check when the page is moved to DASD to determine if the Tag bit is still set.) The Tag Set bit is copied to or from the TLB and handled in a way similar to the reference and change bits of the page.

The AS/400 relies on the MI translator to regulate the generation of Quadword RISC instructions used to define tagged pointers, to move or copy tagged pointers, or to insert or remove the tags from memory. Since the tags are not sent directly to the direct-access storage device (DASD), the tagged pointers have to be extracted from the 4096-byte page by the SLIC code. These tags are then placed separately on DASD and associated with the corresponding page (see section 3.2.3.3.9.6 "Tag Management" on page 3-60). When the page is read from the DASD, the SLIC retrieves the associated tags and causes the tag bits to be set in the page prior to making the page valid to the operating system.

3.1.1.1.3. Hardware Storage Protection

For each page (4096 bytes) read into real memory, the Page Table Entry Page Protect (PP) bits are reserved for storage protection. The processor hardware provides the capability to protect pages of storage as read only, no read or write, or allow read or write. The hardware checks each memory access to verify that the task state allows the operation per the page status in the page table. If not allowed, the memory operation does not complete, and a Data Storage Interrupt or Instruction Storage Interrupt is signaled by the hardware. The same interrupt is used for both page-fault and storage-protection violation. It is the first-level interrupt handler's (see section 3.2.3.8 "SLIC Exception Management" on page 3-96.) responsibility to determine whether the page was not in real memory or if a protection violation occurred. The storage protection bits are also maintained in the TLB so that addresses in the TLB can be checked for storage protection when accessed.

The hardware checks the storage protection status whenever any virtual address storage access is performed. If the machine is in User State (MSR bit 56 = 1), then the page protect bits have the following meaning:

- 00 - No access
- 01 - Read only
- 10 - Read and Write
- 11 - Read only

If the machine is in System State (MSR bit 56 = 0), then the Storage Protect bits have the following definition:

- 00 - Read and Write
- 01 - Read and Write
- 10 - Read and Write
- 11 - Read only

SLIC controls the page access by loading one of the values listed above into the Page Table Entry whenever a virtual address is loaded into the Page Table Entry.

3.1.1.1.4. Multiprocessor Implementation

The architecture of the AS/400 system makes the differences between uni- and multiprocessors relatively minor. The layered architecture makes multiprocessors totally transparent to everything that runs above the MI, i.e., the OS/400 operating system and end users. Multiprocessor operations are primarily at the hardware and SLIC level.

The AS/400 multiprocessors use several functions to guarantee the integrity of the data between processors.

- The Most Favored Request Register (MFRR) is used to signal other processors. When a store is done to the MFRR, an interrupt is generated to the processor indicating that there is a request pending from another processor.

- Cache coherency is maintained between processors and main storage. If one processor changes a cache line, all other copies of the cache line in other processors are invalidated, or, if modified, the line is first copied to main storage. Coherency is implemented via the MESI.

- Software locks are implemented using the RISC instructions LDARX, LWARX, STDCX, and STWCX. These instructions provide a means of loading and storing memory locations atomically.

- All bytes of an aligned double word or quad word Load or Store are atomic in that all bytes of the double word or quad word are read or written at the same instant (not in pieces).

- RISC instructions that order storage accesses SYNC, VSYNC, and EIEIO are used to provide a barrier such that storage operations done before the instruction’s execution are seen before storage operations done following the instruction execution. The RISC instructions SYNC and VSYNC guarantee that Stores or Loads before the instruction execution are seen by another processor before any Load or Store following the RISC instruction SYNC or VSYNC. The RISC instruction EIEIO guarantees only that Stores are seen in the proper order (Loads are not ordered).

- Cache management RISC instructions are provided that need to be broadcast to all processors to invalidate or flush the contents of the cache line in other processors. These instructions allow SLIC to maintain cache coherency when pages are moved to the DASD or when RISC instructions are modified.

- TLB management RISC instructions are provided that need to be broadcast to all processors to invalidate or flush the contents of the TLB in other processors. These instructions allow SLIC to maintain cache coherency when pages are moved to the DASD or when RISC instructions are modified.

3.1.1.2. Input/Output

The I/O devices attached to the AS/400 include DASD, printers, display devices, optical drives and tape drives. These are all non-programmable devices with standard interfaces. They attach to the AS/400 through intelligent controllers called Input/Output Processors (IOPs) which reside on and are managed through the System Products Division (SPD) bus.

3.1.1.2.1. The SPD Bus

The SPD bus has been adopted as the AS/400 bus architecture used to connect the processor or processors to the various I/O processors (see Figure 3-3. “Complex System Configuration Hardware” on page 3-4).

The SPD bus consists of a 32-bit data bus, either copper in the basic system configuration models or optical. The processor unit and each I/O processor are attached to the bus as an I/O bus unit (IOBU). One IOBU is designated the bus control unit (BCU). The BCU is IOBU address 0 and controls arbitration, originates direct selection, and recovers from bus faults. During the early IPL phases, the service processor acts as the BCU. When the SLIC code is loaded and verified, the processor with address 0 takes over as the BCU. Should a processor failure be detected, a signal is sent to the service processor to assume BCU responsibilities.

The bus arbiter is a part of the BCU. The arbiter performs a serial poll of all IOBUs. The serial poll is deactivated at the end of each arbitration cycle. Besides the serial poll for priority, each IOBU can be assigned one of four request priority levels.

Once an IOBU is selected by the poll, it becomes master and begins a bus cycle. The bus cycle operation has two parts. The first is to select a slave IOBU, determine the type of operation, and the direction of data transfer. The second consists of 1 to 16 data cycles for basic storage or unit operations, and up to 1024 data cycles for packet-streaming-mode storage operations. For details about how the bus is used for I/O, see section 3.2.3.7.2 “Bus Transport Mechanism (BTM)” on page 3-85.

3.1.1.2.2. Input/Output Processors

AS/400 moves much of the redundant protocol-dependent I/O processing outboard by providing small specialized processors attached to the SPD bus to control the I/O devices. These are called input/output processors (IOPs). There are two basic families of IOPs. One family controls the workstation display and printer devices, while the other family controls the diskette, DASD, optical, and tape devices. In addition, the Crypto hardware processor and the service processor are attached as IOPs. The purpose of the IOPs is as follows:

- To off-load work from the AS/400 processor(s)
- To provide the capability to attach diverse devices in both hardware and protocols with minimal effect on the processor(s)
- To reduce the logic in the individual I/O device

For details about how the IOPs interact with the processor, see section 3.2.3.7 “SLIC I/O Management” on page 3-79.

The IOPs are loaded with prelinked IBM microcode during initial program load (IPL). There are no interfaces from which a user can modify the current microcode load in the evaluated configuration.[7] The only interfaces are via device drivers to the SPD bus and the connected I/O device. The IOPs are physically contained in the AS/400 enclosure.

3.1.1.2.2.1. Workstation and Printer IOPs

On the evaluated AS/400 hardware, three similar IOP cards are used to attach the workstation and printer devices. A functional diagram of the workings of an IOP is shown in Figure 3-6. “Workstation/Printer IOP” on page 3-13. As seen from this figure, the logic is to interface with the SPD bus and connect to the standard workstation and printer device interfaces. All IOPs use an overlay algorithm to ensure that no residual data is sent either to the program or to the device. The workstation and printer IOP is used to interface with asynchronous (async) devices physically connected by the RS-232 and RS-422 standards, as well as manage display and printer devices physically connected by either twinaxial or twisted-pair cables. They support up to 40 display and/or printer devices each. The microcode functions include keyboard mapping for various national keyboards, ASCII-to-EBCDIC conversion, keystroke management, and error management.

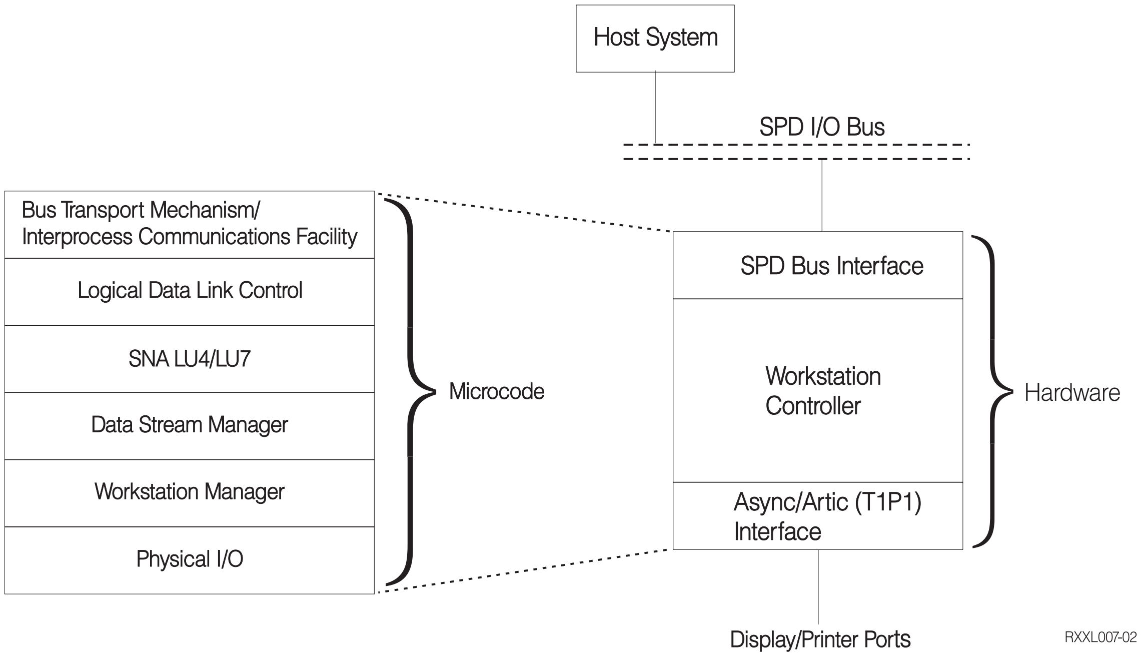


Figure 3-6. Workstation/Printer IOP

The workstations/printers are attached to the controller on a multidrop line with a master-slave relationship. No two devices can have the same address. If a second device attempts to connect with the same address as an existing device, an error occurs and the devices involved must be varied off. In addition, if a logged-on workstation is physically disconnected from the line to the IOP, an error is sent to OS/400 and the active process on the workstation is logged off. Reconnecting or adding a new workstation with the same address requires the user to fully sign on, in the same way as if the user had been logged off.

[7] Administrative users can apply PTFs to modify the microcode, but the system would no longer be in the evaluated configuration.

3.1.1.2.2.2. Diskette, DASD, Optical, and Tape IOPs

The diskette, DASD, optical and tape IOPs consist of two configurations that support the evaluated configuration. One configuration is a multifunction IOP card used for attaching the service processor, as well as for providing storage subsystem services. The other supports only storage subsystem services.

These cards support the interface between the SPD bus and devices that use an IBM implementation of the small computer standard interface (SCSI) or the ANSI 3.8 diskette interface. The SCSI can interleave between SCSI synchronous and asynchronous transfer modes. The microcode architecture consists of an SPD bus manager, bus transfer manager, interprocess communication facility, magnetic media, and error recovery, all running under an IBM-developed control program.

During IPL, the IOP checks the addresses at each port and, if a device is attached, queries the device as to type. During IPL, the Bus Manager examines the bus, identifies all IOPs on the bus, and creates a hardware driver for each IOP. The hardware driver creates the associated Hardware Resource Interface (HRI) Logical and Packaging resource entries and signals events to the OS/400 Device Configuration component requesting that the required configuration object be created. (All IOPs use an overlay algorithm to ensure no residual data is sent either to the program or to the device.)

3.1.1.2.2.3. Service Processor

AS/400 CPUs require a service processor to become operational. Besides initializing the processors, the service processor provides debug and diagnostic functions. The service processor provides control interfaces to the processors and the SLIC software. During IPL, the system is not in the evaluated configuration. The service processor interfaces to the system via the SPD bus and to administrative users through the system control panel. The service processor has a battery-backed clock for maintaining time of day, timed power-on, and automatic-power restart. The logic configuration of the service processor is shown in Figure 3-7. “Service Processor Configuration” on page 3-15. As described above in section 3.1.1.2.2 “Input/Output Processors” on page 3-12, the chip also serves as an IOP with SCSI and communications ports.

The service processor controls the following functions from the control panel:

- Turning system power on or off
- Displaying and reading system reference codes
- Selecting IPL mode functions

Additional information about the service processor and the functions it performs is located in section 4.3.1.4 “What Occurs During an IPL” on page 4-87 and in section 7.2.1 “Power-On/IPL Hardware Diagnostics” on page 7-5.

3.1.1.2.2.4. Crypto Processor

The Crypto processor interfaces to the SPD bus in the same way as other IOPs. The functions of the Crypto IOP are controlled via APIs, which result in I/O commands to the Crypto processor. The primary functions performed by the Crypto processor are data encryption via the Data Encryption Standard (DES) and key management services. The Crypto processor is not used for encrypting AS/400 passwords.

3.1.1.2.2.5. Direct Memory Access Protection

The IOPs may use direct memory access (DMA) to read or write data between main storage and I/O devices. As part of the I/O SLIC support, a lock is obtained on the segments to be read or written and they are pinned in real memory. Since the I/O devices can only access up to 4 GB of main storage with a 32 bit SPD bus address, the 32 bit I/O address is used as a page index into a Translate Control Entry Table with the low-order 12 bits (4 K) used as an offset of the page, and the upper 20 bits are used as an index into the Translate Control Entry Table. When SLIC sends a main storage address to an I/O device to access main storage, SLIC first translates the virtual address to a real main storage address. SLIC then loads this address into the Trans-

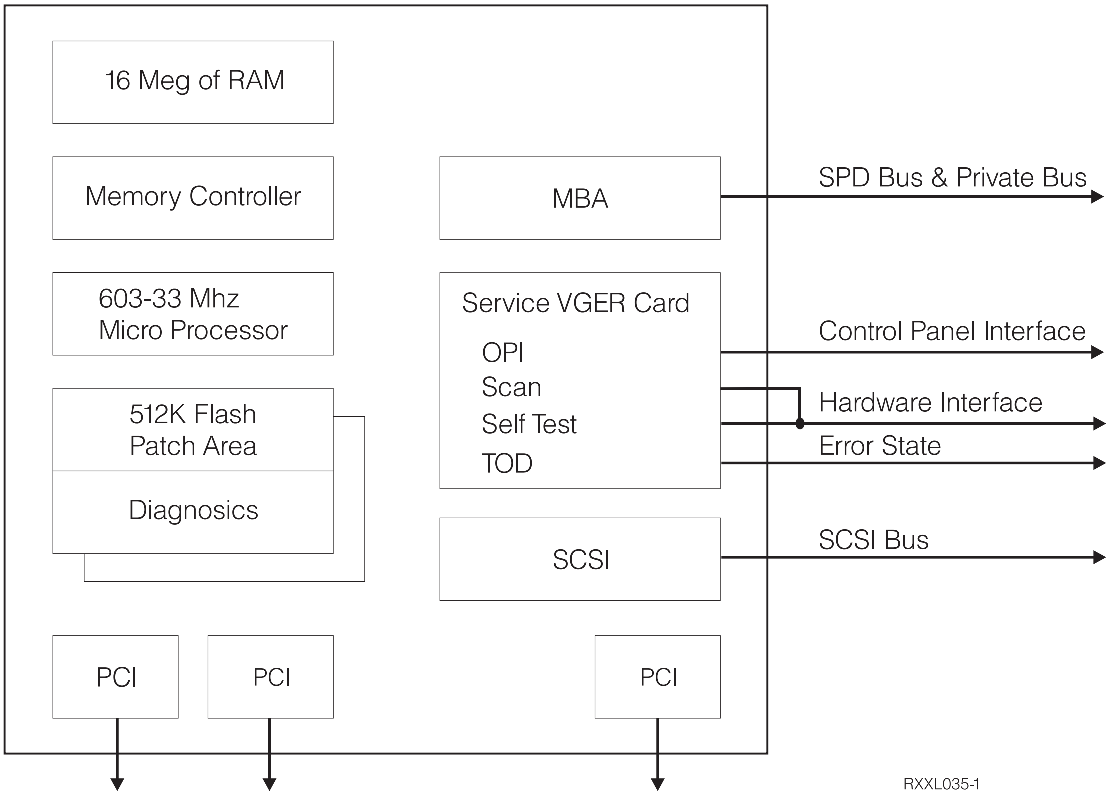


Figure 3-7. Service Processor Configuration

late Control Entry Table using an unused entry in the Table. The index of the entry (Translate Control Entry) used is then placed in a Request Response Control Block (RRCB) that the IOP reads via DMA when notified of a pending command sent to the I/O device along with the 12-bit page offset to form the address for I/O memory accesses. When the I/O device accesses main storage, the I/O device transfers the Translate Control Entry over the SPD bus. The bus chips then use the high 20 bits of the I/O address as an index into the Translate Control Entry Table to locate the real main storage address. A check is made of the Translate Control Entry to determine whether the address is valid and whether the I/O device has write authority to the page. Checks are also made to determine if the I/O address is larger than the Translate Control Entry Table. An I/O address can be used only to reference 4096 bytes of data; however, adjacent I/O pages can be used so that the I/O address space appears to be contiguous while actually referencing main storage in 4 K blocks (virtual I/O address space). SLIC uses a range of 15 Translate Control Entries for each I/O address space request which usually leaves an unused gap in the I/O address range, thus giving an error if the I/O device accidentally increments beyond its own I/O address range.

3.1.1.2.3. Peripheral Devices

A set of peripheral devices consisting of workstations, DASD, diskettes, magnetic tapes, optical devices, and printers can be attached to the AS/400 as needed. These peripherals are all directly attached to the IOPs on the evaluated system by twisted pair, twinaxial, or SCSI cables, and all interfaces to the IOPs are by protocols described previously in the IOP section above. The tapes, DASD, optical devices, and diskettes are housed within the same enclosures as the CPU, while the printers and workstations may be outside of the computer room on multidrop lines.

The peripheral devices in the TCB are all non-programmable (dumb), are all directly attached to an IOP (that is, require no additional controller), and include the following:

- DASD device types
- Diskette device types
- Magnetic tape device types
- Optical storage device types
- Display station device types
- Printer device types

3.2. SLIC and MI

The AS/400 software architecture supports system components at two software layers. For example, program creation is initiated by the OS/400 (above MI) layer and translated by the SLIC layer. This chapter discusses the functions performed in SLIC and the objects on which they are performed.

The MI instruction set includes basic functions commonly needed for programs running on the machine and extended function instructions for monitoring and controlling the operation of the machine. Both user applications and OS/400 use MI instructions.

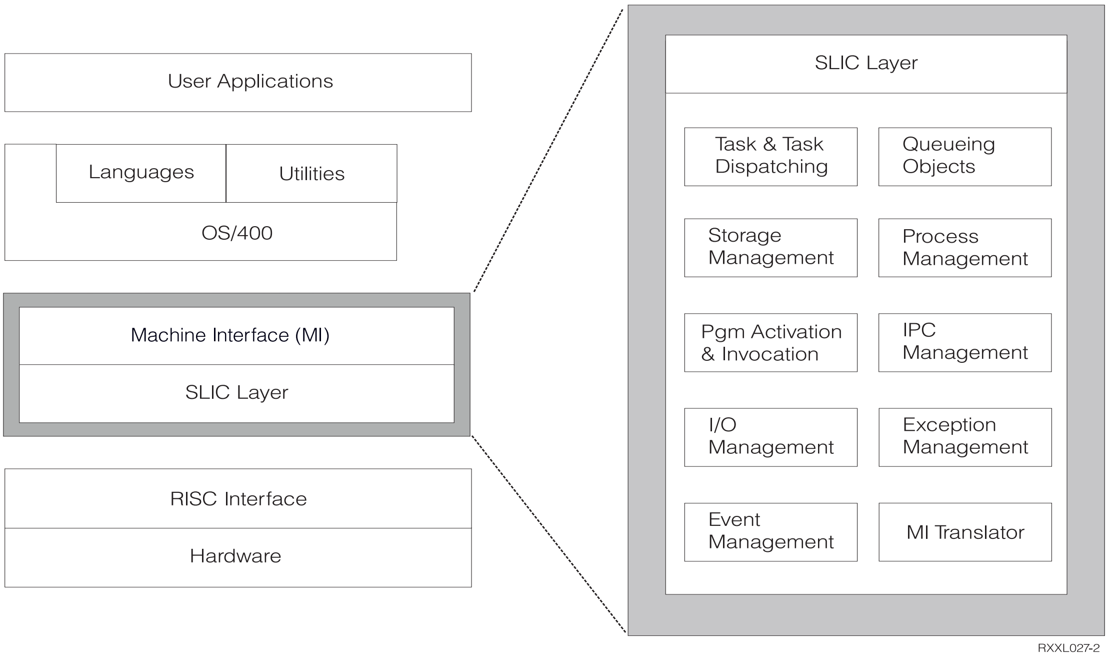


Figure 3-8. The SLIC Layer

Figure 3-8. “The SLIC Layer” on page 3-16 illustrates the software structure of AS/400. The left side of the figure shows the two software layers. The major components of SLIC, expanded to the right side of the figure, and the MI are discussed in detail in this chapter. The objects that are managed are discussed in section 3.2.1 “Object Management in SLIC” on page 3-17. The interfaces to the SLIC are discussed in section 3.2.2 “Machine Interface” on page 3-39. The functions implemented in SLIC are discussed in section 3.2.3 “SLIC Functional Areas” on page 3-43.

3.2.1. Object Management in SLIC

In the SLIC layer, a wide range of MI instructions operate on more than 30 MI object types. Due to restricted interfaces, not all MI object types can be accessed in the evaluated configuration. (Only those available in the evaluated configuration are discussed in section 3.2.1.5 “MI Objects” on page 3-26.)

This section describes the MI object constructs, the MI object types, and management of MI objects.

3.2.1.1. MI Object Format

MI objects, upon which all other objects are based, are building blocks provided by the machine interface to be used by OS/400. An MI object consists of an *encapsulated part* and an optional open or byte-addressable part called an *associated space*. The encapsulated part consists of the *MI object header* and the *functional part*. Most MI objects have the format illustrated in Figure 3-9. “MI Object Format” on page 3-17. Exceptions are described below within the discussions of MI object types.

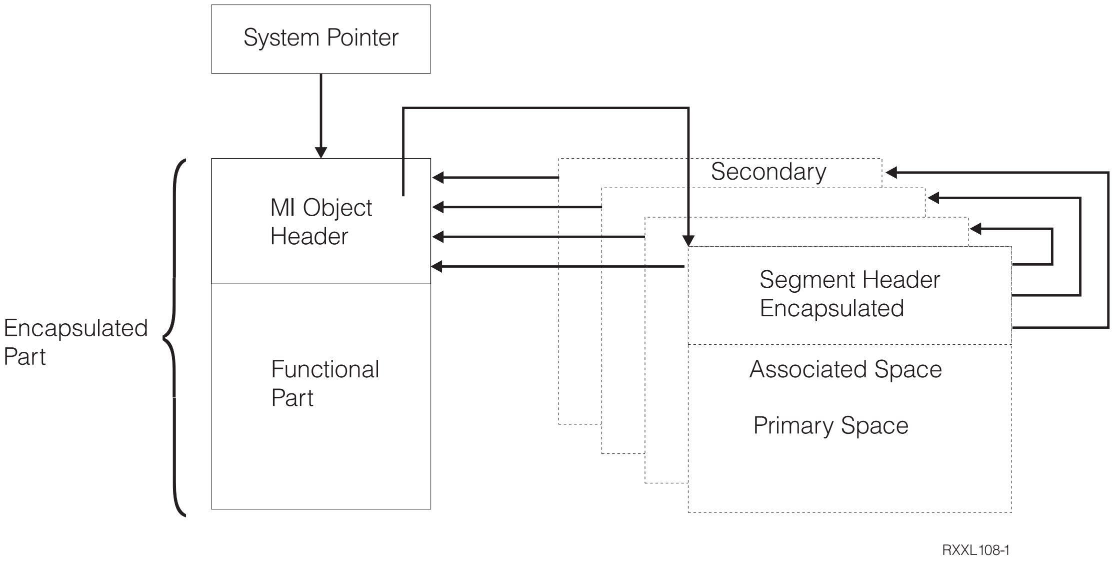


Figure 3-9. MI Object Format

The functional part of an MI object is used to implement a specific function. For example, the functional part of an MI object of type program contains the program’s executable instructions. The functional part is encapsulated, which ensures the integrity of an MI object by not allowing an MI user to modify the encapsulated data, except through MI instructions defined for that purpose, such as the MI instruction Modify Program (MODPG). An MI user can also view some object attributes within the encapsulated functional part by using a specific instruction, such as the MI instruction Materialize Program (MATPG).

The encapsulated part of an MI object also contains the MI object header. The MI object header cannot be directly accessed from above the MI interface, and is viewable only through well-defined interfaces. The first segment of an MI object is referred to as the *base segment*. The base segment always contains the MI object header, which consists of two parts; the *segment header* of the base segment and the *encapsulated program architecture (EPA) header* (both described below).

The encapsulated part of an MI object as shown in Figure 3-9. “MI Object Format” on page 3-17 has the structure depicted in Figure 3-10. “Encapsulated Part” on page 3-18. The segment header of the base segment contains the following information:

- Domain (SYSTEM or USER) of the MI object
- Storage protection class (read/write, read-only from user state, read-only from any state, no access)
- Number of physical pages allocated
- Object base address
- Associated space address

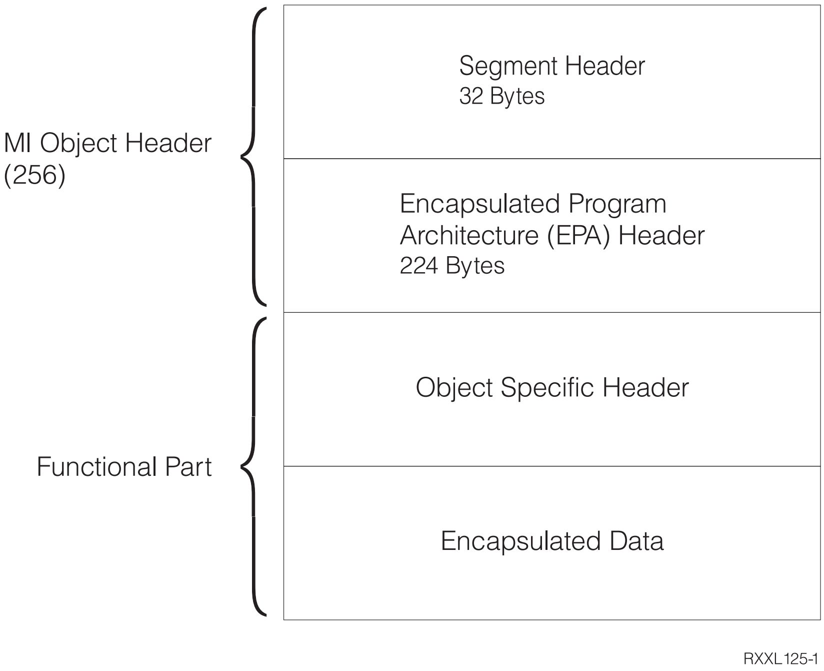


Figure 3-10. Encapsulated Part

The base segment also contains the EPA header. The EPA header contains the following attributes common to all MI objects:

- Name of the object
- Type and subtype of the MI object
- Indicator whether the object is permanent or temporary
- Owning user profile address (this field is blank for temporary objects because no owner exists for temporary objects)
- Owner’s authority (this field is blank for temporary objects)
- Primary group profile address (this field is blank for temporary objects because no primary group exists for temporary objects)
- Primary group authority (this field is blank for temporary objects)
- Public authority (this field is set to *ALL for temporary objects; the system can change it to *EXCLUDE)
- Context address
- Optional pointer to an authorization list for the MI object
- Damage flag
- Object size
- Audit flags list

MI objects occupy one or more segments.[8] However, no one segment is ever part of more than one MI object. MI objects that require more storage than a single segment can provide are stored in multiple segments. Examples of multisegmented objects include programs and large data spaces.

Most MI objects contain an *object-specific header*, which contains properties unique to a specific MI object. For example, the object specific header of a multi-segmented object may contain the pointers to the additional segments. All additional segments point back to the base segment.

A *space* is a contiguous set of bytes that can be addressed down to its individual bytes by a space pointer. If a space is always directly associated with an MI object, it is referred to as an associated space. Other types of spaces may only be associated with an MI object at certain times. For example, the spaces that are used to supply automatic, static, and heap storage for MI programs are associated only with a Process Control Space MI object when the process is active. These spaces are referred to as process spaces instead of associated spaces. They can contain the same data as an associated space, but their attributes cannot be explicitly modified by an MI user. An associated space provides a way to store additional object information. For example, the associated space of a user profile MI object is used to store user attributes for the operating system (e.g., the names of the initial program or the initial screen).

Most MI object types may include one associated space. Some MI object types (e.g., program object) can have more than one associated space, but access group and authorization list MI objects have no associated spaces. Usually an associated space is contained in a segment other than the one containing the encapsulated part of the object. The segment header of the base segment of an object contains addressability to its primary associated space. A header in the primary associated space contains addressability to any and all secondary associated spaces.

[8] MI objects typically comprise three to four segments.

An associated space is created for an MI object using the object creation template. Associating a space with an MI object provides the operating system object manager the capability to store object data in an area addressable through the MI object itself (i.e., once the system pointer to the object is resolved, the object's associated data is available, if the user has object operational authority to the object. See section 3.2.1.2.5 "Resolve System Pointer" on page 3-23).

An associated space can be created fixed or variable in size by specifying the initial allocation in the size of space entry of the object creation template. The maximum size of an associated space is 16 MB.

Associated spaces have the following characteristics:

- Associated spaces can be initially allocated as fixed or variable in size.
- The size of the allocated space is at least as large as requested. When the object size is materialized the space size reflects the current allocation, not the requested size.
- Associated spaces are allocated or extended in multiples of the machine page size (4 K bytes).
- The maximum size of an associated space is dependent upon the type and characteristics of the object containing the space.

3.2.1.2. MI Object Addressability

Addressability to MI objects (except for byte stream files and directories) is usually through contexts. Byte stream file MI objects are addressed through directories.[9] An MI object is made accessible to the user when the object is created and a system pointer is returned by the object-specific create instruction, or when the object's symbolic address is resolved to a system pointer through a context or directory. This section describes how MI objects are addressed symbolically through contexts and directories and referred to by pointers.

3.2.1.2.1. Machine Context

The machine context contains addressability by symbolic address to the following MI objects: user profiles, contexts, logical unit descriptions, and controller descriptions. Addressability to these MI objects cannot be placed into any other context. Unlike a context MI object, the machine context is not an MI object; a user cannot obtain a system pointer to the machine context. It is created at installation time and given a permanent, reserved, virtual address. Addressability to the machine context is via this reserved address.

There is one, and only one, machine context on an AS/400 system.

3.2.1.2.2. Contexts

A context is an MI object that contains addressability by symbolic address to other MI objects. Addressability can be placed in a context, and the context can be used to locate the MI object by its symbolic address (see section 3.2.1.5.6 "Context" on page 3-29). When a context is created, addressability to it is inserted into the machine context.

All MI objects (except those only addressable through the machine context or a directory) are addressed from a temporary context, a permanent context, or no context at all. Most MI objects are first created into "no context", then, once the associated space of the MI object is updated, the MI object is inserted into a context. This method is used to avoid the existence of partially created MI objects in contexts, which may result during a system crash. The operating system component that creates an MI object outside a context is

[9] Addressability to directories is through other directories.

responsible for maintaining addressability to the MI object. The MI instruction Resolve System Pointer (see section 3.2.1.2.5 “Resolve System Pointer” on page 3-23) only operates on MI objects in a context.

3.2.1.2.3. Directories

A directory contains addressability to the following MI objects: directories and byte stream files. Addressability to these MI objects cannot be placed into a context.

A directory is a special form of a data space index MI object. It does not have a separate MI object type.

3.2.1.2.4. MI Pointers

In AS/400, both objects and the data within objects are referred to by pointers.

As shown in Figure 3-11. “Tagged Pointer Structure” on page 3-21, pointers are represented as 16 bytes at the machine interface (MI). Each 16-byte pointer contains:

- **Pointer type**: All pointers are strongly typed. Each MI instruction receiving a pointer as a parameter checks the pointer type field to verify that it is the expected type.
- **Pointer specific information**: Different pointer types may use these bytes to contain information specific to that pointer type. For example, in data pointers some of these bytes are used to define the type and length of data referred to by the data pointer; in system pointers, some of these bytes are used to indicate the number of pages to bring in to main store when some MI system object types are referred to, and to hold an additional authority attribute that doesn't fit in byte 15.
- **Virtual address**: As described in section 3.2.3.3 “Storage Management” on page 3-47, AS/400 supports an 8-byte virtual address space. This field contains the 8-byte virtual address of the associated object or location within the object.

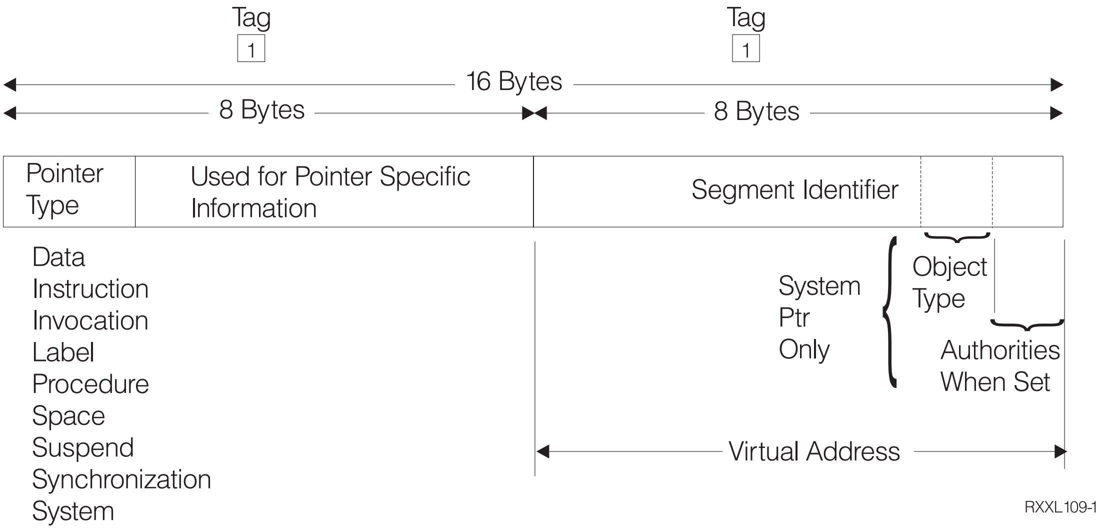


Figure 3-11. Tagged Pointer Structure

To be valid, all pointers that can be addressed at the MI must be tagged. As described in section 3.1.1.1.2 "Tag Bits" on page 3-8, the hardware provides special-purpose support known as a tag bits, which is an additional 2 bits, not addressable above the MI and logically associated with every 16 bytes of main storage. Any attempt to modify a "tagged pointer" (except with the MI instructions designed to operate on that pointer type) causes the tag bits to be cleared.

SLIC and code generated by the translator include a special hardware instruction to load tagged pointers. This instruction detects whether a pointer is tagged and sets a status that can be used to raise an exception or set a null pointer value whenever appropriate. Thus, non-pointer data is never used as a pointer.

The types of pointers used to reference MI objects and locations within spaces are described in more detail in the following sections along with information about the unblocked MI instructions used to set and refer to each type of pointer.

3.2.1.2.4.1. System Pointers

System pointers always address the base segment of an MI object and are used to identify MI objects to the MI instructions that operate on them. Since system pointers address an object by its segment identifier, the offset bytes contained within the virtual address are not used. Therefore, the last two bytes of the address are used to contain the object type and, optionally, the object's authorities. Each MI instruction that uses system pointers examines the object type information to verify that the object is a type on which the MI instruction is defined to operate. When an MI instruction uses a system pointer to an object, it verifies that the process has the needed authority to the object for the requested operation. A system pointer to an object may be obtained by passing the symbolic name of the object as an input parameter to the MI instruction Resolve System Pointer. This unblocked MI instruction is described in section 3.2.1.2.5 "Resolve System Pointer" on page 3-23. A system pointer to an object is also returned by the MI instructions, which create a specific type of MI object.

3.2.1.2.4.2. Space Pointers

Space pointers are used to address bytes within a space. They are the pointer type used by application languages, such as C, to address stack, static, and heap storage, as described in section 3.2.1.5.19 "Process Control Space" on page 3-35. Pointers to the associated space of an MI object are obtained using the unblocked MI instruction Set Space Pointer from Pointer. This MI instruction verifies that the process has object operational (*OBJOPR) authority to the object before returning the space pointer. Once the space pointer is set, SLIC does not enforce access control when the contents of the space are referred to. Therefore, SLIC uses read-only hardware storage protection to protect the associated spaces of sharable user-domain MI objects from being modified by user state programs.

There are two types of space pointers: space pointer data objects and space pointer machine objects. Space pointer data objects are the 16-byte tagged pointers discussed earlier in this section. Space pointer machine objects are not tagged, but cannot be addressed by an MI instruction operand. They can be used in most cases that allow a space pointer data object, except when the address of the space pointer is needed. Space pointer machine objects cannot be counterfeit because they are stored in the Invocation Control Block (ICB), which is an internal stack area. The translator does not allow code to be generated that will take the address of the ICB for an MI instruction operand.

There are unblocked MI instructions available to increment and decrement the offset portion of a space pointer. These MI instructions check that a decrement does not form a negative offset and that an increment does not cause the space pointer to point beyond the maximum allocatable space for the object. These MI instructions raise an exception if they detect an attempt to set a pointer outside this range and will not modify the pointer. Space pointers can never point at the encapsulated portions of MI objects. Consequently, only SLIC code can directly manipulate the encapsulated portions of MI objects.

3.2.1.2.4.3. Data Pointers

Data pointers are a special form of space pointers. The data pointer has an area reserved for the attributes of the data it is pointing to (see Figure 3-11. “Tagged Pointer Structure” on page 3-21) as well as addressability to the space. Data pointer attributes include data type and length information for interpreting the value at the address referenced by the space pointer. Data pointers can be set and manipulated by unblocked MI instructions, similarly to space pointers. The usage of the unblocked MI instructions allows the data type to be declared or changed at runtime as opposed to using the data type declared at compile time. All authority requirements and restrictions of space pointers apply to data pointers.

3.2.1.2.5. Resolve System Pointer

The MI instruction Resolve System Pointer (RSLVSP) locates in a context an MI object identified by a symbolic address and returns a system pointer to the MI object. RSLVSP causes one or more contexts to be searched in order to locate the specified MI object. MI objects not in contexts cannot be resolved with RSLVSP.

The context to be searched can be specified explicitly by symbolic address or it can be provided implicitly. For example, when the MI object type of the object to be resolved can be addressed only by the machine context, the machine context is searched. Also, when a context is not specified as part of the symbolic address of the MI object and the MI object is not of the type addressed only by the machine context, RSLVSP searches the contexts specified in the Name Resolution List (NRL) (see section 3.2.3.4 “Process Management” on page 3-61).

System state programs can also use the MI instruction RSLVSP to set authorities in a system pointer. A system pointer that contains MI object authorities is referred to as an *authorized pointer.*

A process must have pointer authority to set authorities in a system pointer. Also, if the process has pointer authority, then authority can be increased in an authorized pointer. If the process does not have all the requested authorities, then only the requested and available authorities are set in the pointer.

3.2.1.2.6. Directory Look-up

The MI instruction Lookup Directory (LOOKUP) locates in a directory an MI object identified by a name and a specified set of attributes and returns a system pointer. LOOKUP causes only one directory (the specified directory) to be searched in order to locate the specified MI object. MI objects not addressed from a directory cannot be located with LOOKUP. The MI instruction LOOKUP can be used only by system state programs.

The directory to be searched is specified explicitly by a system pointer. Unlike RSLVSP, which searches a list of contexts, LOOKUP searches only the specified directory to locate the MI object.

The MI instruction LOOKUP does not set authorities in the system pointer it returns. However, system state programs can use the MI instruction File Server/System Router (FSROUTE) to set authorities in a system pointer.

3.2.1.3. MI Object Manipulation

Once an MI object address is resolved, or once an MI object is created, the user uses the system pointer for addressability to the MI object. This section describes how an MI object can be created, destroyed, materialized, and modified. This section also describes the significance of MI object domain and MI object persistence.

3.2.1.3.1. Create and Destroy MI Object

Creating and destroying MI objects affects the contexts that contain addressability to the MI objects. Upon MI object creation, addressability can be inserted into a context. Upon MI object deletion, addressability is removed from a context. Also, the MI instruction Modify Addressability causes the MI object addressability contained in a system pointer to be inserted into or transferred to a specific context (if specified) and removed from the context currently addressing the MI object (if any).

Creating and destroying MI objects affects the process user profile. The creating process user profile is assigned ownership of and is given all authorities to the MI object. Temporary MI objects are not owned by any user profile. All object authorities are granted to the public for a temporary MI object.

Each instance of an MI object must be explicitly created via the create instruction provided for that type of MI object. Successful execution of the create instruction returns a system pointer to the MI object. If the MI object is created with an associated space, a space pointer can be obtained from the system pointer.

Each create instruction refers to a template that defines a set of attributes and values to be assigned to the MI object upon its creation. The format of the template is unique for each type of MI object, with each template containing a different set of attribute types. However, each template also contains a set of attributes that are common to all MI objects. These common attributes are as follows:

- MI object identification - provides symbolic identification (object type, subtype, and name) associated with the object and can be used to locate the object through a context.
- Existence - identifies the object as temporary or permanent.
- Space - defines the initial size allocation for the space to be associated with the object.
- Context addressability - identifies a context into which addressability for the object is to be inserted (optional[10]).
- Access group membership (see section 3.2.3.3.2 “Virtual Storage and Single Level Store” on page 3-48) - identifies a group to which the object belongs (temporary objects only).

MI objects are created via the appropriate MI instruction as either temporary or permanent objects. Temporary MI objects are destroyed by the machine at Initial Program Load (IPL), and their addresses are not reusable between IPLs. Also, temporary MI objects can be explicitly destroyed by using the appropriate destroy instruction for the particular type of MI object. A temporary MI object cannot be secured by an authorization list or a private authority. The only authority that can be set for a temporary MI object is *PUBLIC authority.

MI objects are created as either permanent or temporary, depending on whether the MI object is to exist across IPLs. For instance, a user profile is always created as permanent, whereas most job structures are created as temporary. Permanent MI objects must be explicitly destroyed by using the appropriate destroy instruction for the particular type of MI object.

3.2.1.3.2. Materialize MI Object

The only interfaces available to materialize the contents of an MI object are those MI instructions defined for that MI object type. Using the addressability provided by a system pointer, the *materialize* MI instructions (at least one for each MI object type) allow the user at the MI level to view a limited selection of attributes contained within the encapsulated part of an MI object.

For example, the MI instruction Materialize Program returns a copy of many of the components and attributes of an MI program object. The components and attributes are identical to those specified in that MI object's program creation template, unless they have subsequently been modified by the machine or by an

[10] See section 3.2.1.2.2 “Contexts” on page 3-20.

MI user with an appropriate modification instruction. Some of the attributes that are materialized are existence (temporary or permanent), space (fixed- or variable-length), initial value of space, size of space, and context (system pointer to the addressing context). The MI instruction Materialize System Object (MATSOBJ) can be used to retrieve generic attributes (e.g., name and size) of many MI object types.

3.2.1.3.3. Modify MI Object

The *modify* MI instructions (at least one for each MI object type) allow the user at the MI level to change a selection of attributes for a particular MI object type.

Some modify MI instructions can be used more generically to change the attributes of several MI object types. The MI instruction Modify Space Attributes (MODS) can be used to change the size and other attributes (e.g., fixed- or variable-length) of a space MI object or the associated space of any MI object. The MI instruction Modify System Object (MODSOBJ) can be used by OS/400 programs, but not untrusted user programs, to change some generic object attributes, such as a usage timestamp.

3.2.1.3.4. MI Object Domain

MI object types are created in system or user domain. As mentioned above, object domain is indicated in the MI object header. MI objects in the system domain can be accessed only by programs running in system state. MI objects in the user domain can be accessed by either user-state or system-state programs. Some MI objects can be created in either system or user domain, and program objects can be created only in user domain. (Most OS/400 programs, however, are in system domain. They are modified before shipment by an IBM internal tool.) The implementation of MI instructions that accept a system pointer as input perform a domain check by comparing the current execution state against the domain of the object referred to by the system pointer.

3.2.1.4. MI Instruction Blocking

The MI contains a large set of instructions for creating, destroying, accessing, and manipulating MI objects. However, to ensure proper system operation, some of the instructions are “blocked” from being executed or included in user created programs. There are several mechanisms used to block MI instructions.

Translator Blocking
The instruction is not translatable on a customer’s machine. If the instruction is encountered when a program is being translated (converted to a RISC instruction stream), an exception is signaled and the program object is not created.[11]

Runtime Blocking
The instruction is only allowed to execute if the process is currently in “system” state. If the process is in “user” state, an exception is signaled and the instruction is not executed. All runtime blocked instructions are also translator blocked.

Special Environment Blocking
The instruction is only translatable when the MI translator is called via special internal interfaces which are not accessible above the MI (they are only accessible from within SLIC). For example, some MI instructions are only translatable when the MI transformer invokes the MI translator. For more information about the MI transformer and the MI translator see section 3.2.3.10 “MI Translator” on page 3-101.

[11] Translator blocked MI instructions are only translatable by a special version of the MI translator used by IBM to create programs that are part of OS/400.

3.2.1.5. MI Objects

Table 3-2 on page 3-26 lists each MI object type. The table indicates the following for each MI object:

- The managing component (in some cases, a component manages more than one MI object type)
- The domain in which the MI object is created
- Whether the MI object is created temporary or permanent
- Whether the MI object can be created with an associated space and whether the create instruction is blocked

The following MI object types listed in Table 3-2 on page 3-26 do not map to OS/400 external objects. These MI objects are used by OS/400 directly on behalf of users.

- Access group
- Commit block
- Dump space
- Process control space
- Queue space

Table 3-2. MI Object Types

MI Object Type	Managing Component	Domain	Permanent/ Temporary	Created with Associated Space Create Blocked
Access Group	Resource Management	System	Temporary	No-Create is blocked[a]
Authorization List	Authority	System	Permanent	No-Create is blocked
Byte Stream File	Posix	System	Permanent	Yes-Create is blocked
Byte String Space	Byte String	System	Permanent	No-Object type no longer created
Commit Block	Commit	System	Permanent	Yes-Create is blocked
Context	Context Management	System or User (QTEMP)	Temporary (QTEMP) or Permanent	Yes-Create is blocked
Controller Description	Source sink	System	Permanent	Yes-Create is blocked
Cursor	Database	System or User	Either	Yes-Create is blocked
Data Space	Database	System	Either	Yes-Create is blocked
Data Space Index	Database	System	Either	Yes-Create is blocked
Dictionary	Dictionary Service	System	Permanent	Yes-Create is blocked

Table 3-2. MI Object Types

MI Object Type	Managing Component	Domain	Permanent/ Temporary	Created with Associated Space Create Blocked
Dump Space	Load/Dump	System	Either	Yes-Create is blocked
Index	Independent Index	System or User	Either	Yes
Journal Port	Journal	System	Permanent	Yes-Create is blocked
Journal Space	Journal	System	Permanent	Yes-Create is blocked
Logical Unit Description (LUD)	Source Sink	System	Permanent	Yes-Create is blocked
Module	Module Object Manager	System	Either	Yes-Create is blocked
Network Description	Source sink	System	Permanent	Yes-Create is blocked
Process Control Space	Process Management	User	Temporary	Yes-Create is blocked
Program	Program Binder	System or User	Either	Yes-Create is blocked
Queue	Queue Management	System or User	Either	Yes-Create is blocked
Queue Space	Queue Space	System	Either	Yes-Create is blocked
Space	Space Object	System or User	Either	Yes
User Profile	Authority	System	Permanent	Yes-Create is blocked

a. The create instructions are blocked only at the MI level; creation is allowed at the OS/400 level.

The following is a brief description of the management of each of the MI objects.

3.2.1.5.1. Access Group

An access group is an MI object that is a collection of other MI objects, which are transferred to and from auxiliary storage as a group. The access group is used to improve storage management efficiency by specifying which MI objects are used together. An access group is used only by work management, and because it is not associated with an external object, it is not directly accessible to an untrusted user.

The *Resource Management* component manages the access group MI object. The MI instruction Create Access Group creates the access group MI object as temporary, in system domain, and without an associated space. This instruction sets the access group attributes within the object specific header (e.g., pointer to and length of the storage segment).

Addressability to the access group storage segment is contained in the object specific header. The access group storage segment contains a mapping to the addresses of the MI objects belonging to the access group. To be a member of an access group, an MI object must be created in that access group.

In addition to the MI instruction Create Access Group$^{(\textit{blocked})}$, the MI instructions that operate on access-group MI objects are:

- Destroy Access Group$^{(\textit{blocked})}$
- Materialize Access Group Attributes$^{(\textit{unblocked})}$
- Reset Access Group$^{(\textit{blocked})}$
  This instruction also sets the access group attributes within the object-specific header.

3.2.1.5.2. Authorization List

An authorization list is an MI object used to group MI objects together in order to manage the authorizations collectively rather than individually (see section 6.2.3 “Sources of Authority” on page 6-7).

The *Authority* component manages the authorization list MI object. The MI instruction Create Authorization List$^{(\textit{blocked})}$ creates the MI object as permanent and in system domain. This instruction sets the authorization list attributes within the object-specific header (e.g., Ignore private authorities to objects secured by this authorization list).

In addition to Create Authorization List$^{(\textit{blocked})}$, the MI instructions that operate on authorization list MI objects are as follows:

- Destroy Authorization List$^{(\textit{blocked})}$
- Grant Like Object Authority$^{(\textit{blocked})}$
- Materialize Authorization List$^{(\textit{unblocked})}$
  Returns authorization list information and a list of all objects secured by the authorization list

- Materialize Authority$^{(\textit{unblocked})}$
  Refers to the authorization list securing an object in order to get authority for the specified user and public authority

- Modify Authorization List$^{(\textit{blocked})}$
  Secures an object with an authorization list

- Test Authority of the Process$^{(\textit{unblocked})}$
  Refers to the authorization list securing an object to get authority for the process

3.2.1.5.3. Byte Stream File

A byte stream file is an MI object used to represent the stream file, *STMF, object. The byte stream file is used to store user data.

The *POSIX* component manages the byte stream file MI object. The MI instruction File Server/System Router$^{(\textit{blocked})}$ creates the MI object as permanent and in system domain. This instruction sets the byte stream file object attributes within the object-specific header.

In addition to File Server/System Router$^{(\textit{blocked})}$, the MI instructions that operate on auxiliary server MI objects are as follows:

- File System Read$^{(\textit{blocked})}$
- File System Write$^{(\textit{blocked})}$

3.2.1.5.4. Byte String Space

A byte string space is an obsolete MI object, which cannot be created on RISC systems. It was available on IMPI systems for trusted users (i.e., all byte string space instructions were blocked and byte string space objects were created in system domain.) Byte string space objects can exist on a RISC system that was upgraded from IMPI, even though byte string spaces can no longer be created.

The only MI instructions that operate on byte string space MI objects are as follows:

- Destroy Byte String Space(*blocked*)
- Materialize Byte String Space(*blocked*)

3.2.1.5.5. Commit Block

A commit block is an MI object used by database management to hold information concerning the changes made to objects under commitment control. The commit block contains a list of the objects under commitment control, a list of data-space record-lock identifiers, a list of objects that have undergone changes in the current commit cycle, and the commit ID.

The *commit* component manages the commit block MI object. The MI instruction Create Commit Block(*blocked*) creates the MI object as permanent and in system domain.

In addition to Create Commit Block(*blocked*), the MI instructions that operate on commit block MI objects are as follows:

- Commit(*blocked*)
- Decommit(*blocked*)
- Destroy Commit Block(*blocked*)
- Materialize Commit Block Attributes(*unblocked*)
- Modify Commit Block(*blocked*)

3.2.1.5.6. Context

Context MI objects contain addressability to other MI objects. A context object relates the name of the MI object found in the object header to the address of that object. A context object is used in system pointer resolution to obtain system pointers to MI objects. Certain types of MI objects, including context MI objects, can be addressed only by the machine context (see section 3.2.1.2.1 “Machine Context” on page 3-20).

A context contains addressability by symbolic address (type, subtype, and name) to other MI objects. If addressability is placed in a context, then the context can be used to locate the MI object by its symbolic address. The symbolic address of every object addressed by a single context must be unique (i.e., two objects with the same name, type, and subtype cannot be addressed by the same context). A maximum of one context can be used to address an MI object.

The AS/400 system can specify a context to initially address the MI object in the context addressability attribute of the MI object template. Alternatively, if a context is not specified by the user (e.g., temporary space and temporary index MI objects created in user domain), addressability is not placed in any context. The OS/400 user may subsequently insert MI object addressability into a context, move MI object addressability from one context to another, or delete MI object addressability from a context. A system pointer is obtained from a context using a resolve instruction.

The *Context Management* component manages the context MI object. The MI instruction Create Context (CRTCTX)(*blocked*) explicitly creates a permanent or temporary system context.

Each job on the AS/400 has a *QTEMP* context associated with it. QTEMP is created via the MI instruction CRTCTX, as are all other contexts. However, the QTEMP context is created as a temporary object to allow a user application (running in user state) to create temporary objects directly into the QTEMP context via the MI instruction *Create Space*. These space objects are used by the application for temporary storage. Since all other contexts are created as system domain objects, a user cannot create an object directly into the context via the MI instruction but must use the TCB-provided interfaces for creating MI objects.

Since QTEMP is created as a temporary object, addressability to QTEMP is not placed in the machine context. Since the QTEMP address is not in the machine context, then no other job on the system (running in user state) can get access to another job's QTEMP. (RSLVSP will not find a QTEMP context.) A pointer to the QTEMP context for the job is stored by the operating system in the job structure's work control block.

Objects can be created into the QTEMP context by using CL commands or APIs that create an object (e.g., the CL command Create Physical File [CRTPF]). When an object is created by the TCB interface (CRTPF in this case), the object is created as a permanent object, although it is created into the QTEMP context. This allows for the object to be moved from the QTEMP context to another context on the system. Any objects that exist in the QTEMP context when the job ends are destroyed by the operating system when the job terminates.

CRTCTX sets the context-specific attributes within the object-specific header. The object specific header also contains a pointer to an optional Changed Object List of the objects that have changed since a given time.

The index portion is contained in the encapsulated functional part of the context MI object. The index portion contains one object-addressability entry for each object addressed by the context.

In addition to Create Context$^{(\mathit{blocked})}$, the MI instructions that operate on context MI objects are as follows:

- Destroy Context$^{(\mathit{blocked})}$
  Objects must be empty before the context is destroyed.
- Modify Addressability$^{(\mathit{blocked})}$
  Explicitly inserts object addressability into a context and/or explicitly deletes object addressability from a context. However, addressability cannot be deleted by this instruction if the object is addressed by the machine context. Modify Addressability causes the object addressability indicated by a system pointer to be inserted into or transferred to a specific context (if specified) and removed from the context currently addressing the object (if any).
- Materialize Context$^{(\mathit{unblocked})}$
  Materialize all or a subset of the context entries. For each object selected, a system pointer addressing the object and/or the object's symbolic address can be materialized.
- Modify Context $^{(\mathit{blocked})}$
- Rename Object$^{(\mathit{blocked})}$
- Resolve System Pointer$^{(\mathit{unblocked})}$

3.2.1.5.7. Controller Description

A controller description is an MI object that represents either an I/O controller for a cluster of I/O devices or a station that attaches groups of communications devices over the same data link.

The *Source Sink* component manages the controller description MI object. The MI instruction Create Controller Description$^{(\mathit{blocked})}$ creates the MI object as permanent, in system domain, and with an associated space.

In addition to Create Controller Description$^{(\mathit{blocked})}$, the MI instructions that operate on controller description MI objects are as follows:

- Destroy Controller Description$^{(\mathit{blocked})}$
- Materialize Controller Description$^{(\mathit{unblocked})}$
- Modify Controller Description$^{(\mathit{blocked})}$

3.2.1.5.8. Cursor

A cursor is an MI object used to provide access to the entries residing within a data space. A cursor is the system's only interface to these entries.

A cursor can directly locate any entry in a data space; a cursor can also indirectly locate any entry in a data-space through a data space index. A cursor, then, serves as the interface for retrieving data from and storing data into the data space (see section 4.2 "Database Management System" on page 4-68).

The *Database* component manages the cursor MI object. The MI instruction Create Cursor$^{(\textit{blocked})}$ creates the MI object as permanent or temporary, in system or user domain, and with or without an associated space.

The permanent cursor object specific header contains addressability to one or more data space MI objects and addressability to an optional data space index MI object. Because the data space, data space index, and cursor have addressability to one another, the data space and data space index MI instructions can refer to or modify the cursor MI object.

In addition to Create Cursor$^{(\textit{blocked})}$, the MI instructions that operate on cursor MI objects are as follows:

- Activate Cursor$^{(\textit{blocked})}$
- De-activate Cursor$^{(\textit{blocked})}$
- Destroy Cursor$^{(\textit{blocked})}$
- Materialize Cursor Attributes$^{(\textit{unblocked})}$
- Set Cursor$^{(\textit{blocked})}$
- Retrieve Data Space Entry$^{(\textit{blocked})}$
- Retrieve Sequential Data Space Entries$^{(\textit{blocked})}$
- Retrieve Simple Data Space Entries$^{(\textit{blocked})}$
- Release Data Space Entry$^{(\textit{blocked})}$

3.2.1.5.9. Data Space

A data-space MI object is the basic unit of storage for a database user's data. Database data is contained in the functional part of the data space and consists of a collection of entries. An entry consists of a non-data portion and a data portion. The non-data portion contains information about the entry (e.g., the record is deleted). The data portion contains the actual data (i.e., record) (see section 4.2.1.3 "AS/400 Database File Structure" on page 4-76).

The *Database* component manages the data-space MI object. The MI instruction Create Data Space$^{(\textit{blocked})}$ creates the MI object as permanent or temporary, in system domain, and with or without an associated space.

The object-specific header contains the address of a database cursor MI object (see above) and, optionally, one or more data space index MI objects (see below). Database data is contained in the functional part.

In addition to Create Data Space(*blocked*), the MI instructions that operate on data-space MI objects are as follows:

- Copy Data Space Entry(*blocked*)
- Create Data Space Image(*blocked*)
- Delete Data Space Entry(*blocked*)
- Destroy Data Space(*blocked*)
- Ensure Data Space Entries(*blocked*)
- Insert Data Space Entry(*blocked*)
- Insert Sequential Data Space Entries(*blocked*)
- Materialize Data Space Attributes(*blocked*)
- Modify Data Space Attributes(*blocked*)
- Update Data Space Entry(*blocked*)
- Various Maintenance Functions (DBMAINT)(*blocked*)

3.2.1.5.10. Data Space Index

A data-space index is a MI object based on a *machine index* that is used to logically order the entries in a data space. A machine index provides a means for locating a record, a value, or a function. A machine index provides a search mechanism for finding specific entities and groups of related entities. Portions of the index can be grouped together into logical pages so that highly referenced portions of the index tend to remain resident in main storage.

Through use of a data-space index, the entries in a data space can be accessed independently of the physical organization of the data. A data-space index can be created for one or more data spaces for which a relationship can be expressed through a collection of key fields.

A special form of the data space index MI object is used for directory MI objects.

The *Database* component manages the data space index MI object. The MI instruction Create Data Space Index(*blocked*) creates the MI object as permanent or temporary and in system domain.

The object-specific header has addressability to one or more data space MI objects.

In addition to Create Data Space Index(*blocked*), the MI instructions that operate on data space index MI objects are:

- Estimate Size of Data Space Index Key Range(*blocked*)
- Destroy Data Space Index(*blocked*)
- Materialize Data Space Index Attributes(*blocked*)
- Modify Data Space Index Attributes(*blocked*)

When a data-space index is used as a directory, the following additional MI instructions operate on the data-space index MI object:

- Link Directory(*blocked*)
- Lookup Directory(*blocked*)
- Materialize Object Links(*blocked*)
- Modify Directory Entry(*blocked*)
- Rename Directory Object(*blocked*)
- Unlink Directory(*blocked*)

3.2.1.5.11. Dictionary

A dictionary is an MI object that serves as a storage area for a list of words and supports word verification functions. Some dictionary objects are shipped with the machine and provide for word verification in several different languages (e.g., a dictionary can be used to check spelling, synonyms, hyphenation, and reading grade level).

The *Dictionary Service* component manages the dictionary MI object. The MI instruction Create Dictionary$^{(\textit{blocked})}$ creates the MI object as temporary or permanent and in system domain. This instruction (and others indicated below) sets the dictionary attributes within the object specific header.

In addition to Create Dictionary$^{(\textit{blocked})}$, the MI instructions that operate on dictionary MI objects are as follows:

- Check Dictionary$^{(\textit{blocked})}$
- Destroy Dictionary$^{(\textit{blocked})}$
- Materialize Dictionary Attributes$^{(\textit{blocked})}$
- Search Dictionary$^{(\textit{blocked})}$

3.2.1.5.12. Dump Space

A dump space is an MI object that serves as a storage area for a dump of other MI objects. It provides an on-line storage alternative to the commonly used off-line storage media for dumps of MI objects. The dumping mechanism copies everything from a single MI object to multiple MI objects into the same dump space.

The *Load/Dump* component manages the dump space MI object. The MI instruction Create Dump Space$^{(\textit{blocked})}$ creates the MI object as permanent and in system domain. This instruction (and others indicated below) sets the dump space attributes within the object specific header.

In addition to Create Dump Space$^{(\textit{blocked})}$, the MI instructions that operate on dump space MI objects are as follows:

- Destroy Dump Space$^{(\textit{blocked})}$
- Materialize Dump Space$^{(\textit{unblocked})}$
- Insert Dump Data$^{(\textit{blocked})}$
  Sets dump space attributes such as the dump space size
- Modify Dump Space$^{(\textit{blocked})}$
  Sets dump space attributes such as the dump space size, dump data size, and dump data size limit
- Retrieve Dump Data$^{(\textit{blocked})}$
- Request Path Operation$^{(\textit{blocked})}$
  Performs load and dump functions and sets dump space attributes

3.2.1.5.13. Index

An index is an MI object based on a machine index that is used to store and retrieve data based on a key value. Scalar values and pointers can be inserted into the index with a portion of the value interpreted as the key field. The key value can subsequently be used to locate and retrieve one or more entries from the index.

The *Independent Index* component manages the index MI object. The MI instruction Create Independent Index$^{(\textit{blocked})}$ creates the MI object as temporary or permanent and in system or user domain. This instruction (and the modify instruction indicated below) sets the index attributes within the object specific header.

In addition to Create Independent Index$^{(\textit{unblocked})}$, the MI instructions that operate on index MI objects are as follows:

- Destroy Independent Index$^{(\textit{unblocked})}$
- Find Independent Index Entry$^{(\textit{blocked})}$
- Insert Independent Index Entry$^{(\textit{blocked})}$
- Materialize Independent Index Attributes$^{(\textit{unblocked})}$
- Modify Independent Index$^{(\textit{blocked})}$
- Remove Independent Index Entry$^{(\textit{blocked})}$

3.2.1.5.14. Journal Port

A journal port is an MI object used to link MI objects to journal spaces. Only those MI objects having their changes journaled are linked to a journal space. A journal port contains the definition of the prefix data associated with each change entry in the journal space.

The *Journal* component manages the journal port MI object. The MI instruction Create Journal Port$^{(\textit{blocked})}$ creates the MI object as permanent and in system domain.

In addition to Create Journal Port$^{(\textit{blocked})}$, the MI instructions that operate on journal port MI objects are as follows:

- Apply Journaled Changes$^{(\textit{blocked})}$
- Destroy Journal Port$^{(\textit{blocked})}$
- Journal Data$^{(\textit{blocked})}$
- Materialize Journaled Object Attributes$^{(\textit{unblocked})}$
- Materialize Journaled Objects$^{(\textit{unblocked})}$
- Materialize Journal Port Attributes$^{(\textit{unblocked})}$
- Modify Journal Port$^{(\textit{blocked})}$
- Retrieve Journal Entries$^{(\textit{blocked})}$

3.2.1.5.15. Journal Space

A journal space is an MI object used to contain the changes to those objects specified as journaled objects. When a journal space is attached to a journal port, all changes that occur to the MI objects attached to that journal port are sequentially entered in the journal space. These change entries are of variable length.

The operating system inserts entries in the journal space by using the MI instruction Journal Data. The entries are read using the MI instruction Retrieve Journaled Entries.

The *Journal* component manages the journal space MI object. The MI instruction Create Journal Space$^{(\textit{blocked})}$ creates the MI object as permanent and in system domain.

In addition to Create Journal Space$^{(\textit{blocked})}$, the MI instructions that operate on journal space MI objects are as follows:

- Destroy Journal Space$^{(\textit{blocked})}$
- Materialize Journal Space Attributes$^{(\textit{unblocked})}$

3.2.1.5.16. Logical Unit Description

A logical unit description (LUD) is an object that represents a physical I/O device such as a printer.

The *Source Sink* component manages the logical unit description MI object. The MI instruction Create Logical Unit Description(*blocked*) creates the MI object as permanent and in system domain with an associated space.

In addition to Create Logical Unit Description(*blocked*), the MI instructions that operate on logical unit description MI objects are as follows:

- Destroy Logical Unit Description(*blocked*)
- Materialize Logical Unit Description(*unblocked*)
- Modify Logical Unit Description(*blocked*)
- Request I/O(*blocked*)
- Request Path Operation(*blocked*)

3.2.1.5.17. Module

Module objects are used to create a bound program object. The module objects are bound together to create an executable program object. Module objects are used only in the ILE environment and are not directly callable. They must first be bound together to form the program. The program is then callable.

The *Module Object Manager* component manages the module MI object. The MI instruction Create Module(*blocked*) creates the MI object as temporary or permanent and in system domain.

In addition to Create Module(*blocked*), the MI instructions that operate on module MI objects are as follows:

- Destroy Module(*blocked*)
- Materialize Module(*blocked*)
- Modify Module(*blocked*)

3.2.1.5.18. Network Description

A network description is an object that represents a physical I/O port and communications line for remotely attached I/O devices.

The *Source Sink* component manages the network description MI object. The MI instruction Create Network Description(*blocked*) creates the MI object as permanent and in system domain with an associated space.

In addition to Create Network Description(*blocked*), the MI instructions that operate on network description MI objects are as follows:

- Destroy Network Description(*blocked*)
- Materialize Network Description(*unblocked*)
- Modify Network Description(*blocked*)

3.2.1.5.19. Process Control Space

A process control space is an MI object that is used as part of process execution. A process control space must be created before a process can be initiated. The process control space is then associated with that process for the entire period that the process exists. The machine uses the process control space for machine work areas and storage related to process execution (see section 3.2.3.4 “Process Management” on page 3-61 for a discussion of process management).

The *Process Management* component manages the process control space MI object. The MI instruction Create Process Control Space(*blocked*) creates the MI object as temporary and in user domain.

In addition to Create Process Control Space(*blocked*), the MI instructions that operate on process control space MI objects are as follows:

- Destroy Process Control Space(*blocked*)
- Initiate Process(*blocked*)
- Materialize Process Attributes(*unblocked*)
- Modify Process Attributes(*blocked*)
- Suspend Process(*blocked*)
- Resume Process(*blocked*)
- Terminate Instruction(*blocked*)
- Terminate Process(*blocked*)

3.2.1.5.20. Program

A program is an MI object that forms the basic executable unit of the machine. The execution of a program causes a series of functions to be performed against a set of objects. A program is the encapsulated and executable form of a program template and logically contains both the function definition and object definitions from the program template.

The *Program Binder* component manages the program MI object. The MI instructions Create Program(*blocked*) and Create Bound Program(*blocked*) create the MI object as temporary or permanent and in user domain.

Program *observability* is the attribute used to indicate that a program contains the program template information that was input to the translator (see section 3.2.3.10 “MI Translator” on page 3-101). Program observability provides the administrator the capability to retranslate the program and the capability to use the debug tools for the program.

In addition to the create instructions, the MI instructions that operate on program MI objects are as follows:

- Delete Program Observability(*blocked*)
- Destroy Program(*blocked*)
- Extract Module(*blocked*)
- Materialize Bound Program(*unblocked*)
- Materialize Program(*unblocked*)
- Modify Bound Program(*blocked*)
- Modify Program(*blocked*)

3.2.1.5.21. Queue

A queue is an MI object that can be used for storage and retrieval of data. Any jobs with authority to the queue can store and retrieve data from that queue. This makes a queue convenient for communications between programs. A process can test for data on a queue and either wait or continue execution if the data are not available. Data can be inserted on and removed from a queue based on a key value, or they can be processed based on order of arrival, either first-in-first-out (FIFO) or last-in-first-out (LIFO).

The *Queue Management* component manages the queue MI object. The MI instruction Create Queue(*blocked*) creates the MI object as temporary or permanent and in system or user domain.

In addition to Create Queue$^{(\textit{blocked})}$, the MI instructions that operate on program MI objects are as follows:

- Dequeue$^{(\textit{unblocked})}$
- Destroy Queue$^{(\textit{blocked})}$
- Enqueue$^{(\textit{unblocked})}$
- Materialize Queue Attributes$^{(\textit{unblocked})}$
- Materialize Queue Messages$^{(\textit{unblocked})}$

3.2.1.5.22. Queue Space

The *Queue Space* component manages the queue space MI object. The MI instruction Create Queue Space$^{(\textit{blocked})}$ creates the MI object as temporary or permanent and in system domain.

In addition to Create Queue Space$^{(\textit{blocked})}$, the MI instructions that operate on process queue space MI objects are as follows:

- Destroy Queue Space$^{(\textit{blocked})}$
- Materialize Process Message$^{(\textit{unblocked})}$
- Modify Process Message$^{(\textit{blocked})}$
- Materialize Queue Space Attributes$^{(\textit{blocked})}$
- Modify Queue Space Attributes$^{(\textit{blocked})}$
- Send Process Message$^{(\textit{blocked})}$

3.2.1.5.23. Space

A space object contains only a space. It is used to store scalars and pointers.

The *Space Object* component manages the space MI object. The MI instruction Create Space$^{(\textit{unblocked})}$ creates the MI object as temporary or permanent and in user or system domain.

In addition to Create Space$^{(\textit{unblocked})}$, the MI instructions that operate on space MI objects are as follows:

- Destroy Space$^{(\textit{unblocked})}$
- Materialize Space Attributes$^{(\textit{unblocked})}$
- Modify Space Attributes$^{(\textit{unblocked})}$

3.2.1.5.24. User Profile

A user profile is the MI object that identifies a valid user to the machine. A user profile contains the addresses of MI objects to which it has private authority and contains those private authorities. A user profile defines the user's machine resource authorization (amount of system resource available to the user) and operational attributes (e.g., name of the print queue where output is saved or the name of the initial program menu that determines what options are available to the user). A user profile also contains the special authorities necessary to execute some set of instructions, and the special privileged instruction mask (see section 5.2.6 "User Classes and Special Authorities" on page 5-4).

As an option, a user profile can identify a group profile (a special use of the user profile object) that is used as a source of security attributes. Users can have from zero to 16 group profiles but group profiles cannot have a group profile.

The *Authority* component manages the user profile MI object. The MI instruction Create User Profile$^{(\textit{blocked})}$ creates the user profile MI object as permanent, in system domain, and with an associated space.

This instruction, as well as the MI instruction Modify User Profile$^{(blocked)}$, sets the user profile attributes within the object- specific header. These attributes include the security relevant attributes special authorities, special privileged instruction mask, and auditing fields.

The index portion is contained in the encapsulated functional part of the user profile MI object. This index portion is implemented as a machine index construct. The index portion contains five index entry types:

- Owned object entry for each owned object
- Entry for each object for which this user profile is the primary group.
- Authorized object entry for each object (owned by another user profile) that the user profile has been granted authority to use
- Authorized user entry for each owned object for which authority has been granted to another user
- Free entry

In addition to Create User Profile$^{(blocked)}$ and Modify User Profile$^{(blocked)}$, the MI instructions that operate on user profile MI objects are as follows:

- Destroy User Profile$^{(blocked)}$
- Materialize Authorized/Owned Objects$^{(unblocked)}$
- Materialize User Profile$^{(unblocked)}$

3.2.1.6. Locks

Below the MI, SLIC provides the underlying seize support on a specific machine address to serialize conflicts among users who want the same resource. Above the MI, SLIC provides MI locking support to serialize conflicts on MI objects, space locations, and object locations. SLIC uses hold records, which represent these seizes or MI locks. Each time a seize or MI lock request is successfully issued, a hold record is created that contains the TDE address, object machine address, the type of hold, and chaining information, which facilitates access to related hold records. The hold record structures are managed as a hash table, whose access is available only by SLIC.

MI objects can be synchronized through the use of *locks*, which can be obtained within programs. Locks are not MI objects. The Resource Management component is responsible for managing locks. Locks are enforced based on the operation to be performed on the MI object. There are five lock states:

- Lock exclusive no read (LENR)
- Lock exclusive allow read (LEAR)
- Lock shared update (LSUP)
- Lock shared read only (LSRO)
- Lock shared read (LSRD)

Locks are not required in order for an object to be referred to (or modified) by an instruction. However, a process is not allowed to use an object if any other process holds a conflicting lock.

The MI instruction Lock Object$^{(unblocked)}$ uses a *lock request template* to request locks on MI objects. The lock request template includes the system pointers of MI objects and the desired lock state for each. A user process requires the following to lock an MI object:

- Ownership of the MI object or any one type of authority
- Execute authority on the contexts referred to for address resolution

The following MI instructions require only execute authority on the contexts referenced for address resolution:

- Unlock Object(*unblocked*)
- Transfer Object Lock(*unblocked*)
- Materialize Selected Locks(*unblocked*)
- Materialize Process Record Locks(*unblocked*)
- Materialize Process Locks(*unblocked*)
- Materialize Object Locks(*unblocked*)
- Materialize Data Space Record Locks(*unblocked*)

The following MI instructions require no authority to execute once a space pointer is obtained:

- Lock Space Location(*unblocked*)
- Materialize Allocated Object Locks(*unblocked*)
- Unlock Space Location(*unblocked*)

The following MI instructions require no authority to execute once an Object pointer is obtained:

- Lock Object Location(*unblocked*)
- Unlock Object Location(*unblocked*)

Each execution of a lock MI instruction causes the *Lock Table* to be updated. The Lock Table resides in SLIC storage with no interfaces to it from above the MI. The Lock Table contains all current information on MI objects that are locked or requesting locks.

3.2.2. Machine Interface

The Machine Interface (MI), is the interface between the SLIC layer of AS/400 and the OS/400 and application layers, as depicted in Figure 3-8. “The SLIC Layer” on page 3-16. The MI interface provides an object-based instruction set to operate on MI objects, which may be directly invoked by users. The MI instruction set is the lowest level interface available to OS/400 and applications and can be viewed as the machine instruction set that typically makes up assembly programs. However, MI instructions are not directly executable on the hardware and must be translated into hardware instructions (some of which may call SLIC).

There are two variations of the MI, the New Machine Interface (NMI) and the Original Machine Interface (OMI). The programmer does not explicitly choose which variation to use; the choice is based on the compiler that is used to generate the MI program template. See section 3.2.3.10.1 “OMI Instructions” on page 3-102 for a description of OMI and NMI. The MI template is used by the translator and contains the MI instructions that are to be translated to hardware instructions. In the OMI architecture, the operands are specified in the instruction stream along with the opcode. In the NMI architecture, the NMI instruction contains only the opcode; the operands are conceptually located on a stack. The NMI architecture also introduces a different set of data types and instructions. The NMI allows users to create bound programs, which use the binder to resolve references between modules to form an executable object. The capability to create modules and bind them to other modules is not available in OMI. The translator creates the instruction streams that are incorporated into programs by the program binder (binder). However, translation of OMI templates requires a preliminary step to transform the OMI template into a NMI template. See section 3.2.3.10 “MI Translator” on page 3-101 for more information.

In addition to the normal performance concerns associated with a code generator, the AS/400 translator must also provide support for system integrity. It does this by producing all the executable code for each MI program, in accordance with the MI instruction stream specified. Some correctness checks, such as stack operand types, are made at translation time. Run-time checks, such as verifying tagged pointer uses, are included by the translator in the code it generates.

Not all MI instructions available for OS/400 can be used by customer programs. The translator prevents these instructions from being included in customer programs. See section 3.2.3.10 “MI Translator” on page 3-101 for more information about how these instructions are restricted from untrusted users. In addition to the translator checks performed at translation time, checks are also made at run-time to ensure that users cannot execute programs that contain blocked instructions while their process is in user state. See section 3.2.3.10.6 “The Router” on page 3-105 for more information about this run-time checking. Additionally, there are instructions that require the user to have a special authority. These are called special privileged instructions and are checked at run-time as well.

3.2.2.1. MI Characterization

The MI provides a powerful interface that includes both simple and complex operations. Simple instructions may be used for copying bytes of data but a single MI instruction can also be used to initiate a process or to call an MI program. Thus, the implementations of some MI instructions are responsible for verifying parameters and making security-related checks and decisions (i.e., discretionary access control (DAC) checks, domain checks, switching the process state, controlling certain functionality, and generating audit records). Instructions that perform security-relevant events also have the ability to generate audit records. The security-relevant aspects of the MI are discussed in the appropriate section of this report (e.g., DAC, Audit, State Transition - section 3.2.3.5.3 “State Transition” on page 3-71).

The ability to provide a secure interface is predicated upon control of addressing. MI allows pointers to be created and manipulated only in well-defined ways; random virtual addresses cannot be used. The implementations of all MI instructions that work with pointers verify pointer validity.

3.2.2.2. MI Instruction Set

As stated earlier, the MI instruction set is object based and provides instructions to operate on MI objects. This section characterizes the MI instruction set and provides a description of the generic functionality it provides.

In addition to the checks provided by hardware instructions, MI instructions (those that take system pointers as parameters) perform DAC checks and are capable of generating audit records.

3.2.2.2.1. Blocked Instructions

Blocked MI instructions are those that must not be translated into executable code by the translator. IBM has an internal translator that translates blocked instructions, but the translator that is shipped to customers prevents the translation of blocked MI instructions. See section 3.2.3.10 “MI Translator” on page 3-101 for more information about how this is controlled.

3.2.2.2.2. Partially Blocked

Several MI instructions (e.g., Create Space, Destroy Space, Modify Space) are unblocked at translation time, but are partially blocked at run-time while the machine is in the evaluated configuration. Partially blocked means that not all of the defined functionality for that instruction is available. These MI instructions check the state of the process if particular input values are used and, if the process is in user state, an exception is raised.

3.2.2.2.3. Unblocked Instructions

The unblocked MI instructions are an interface provided to the untrusted user. As stated earlier, even unblocked MI instruction implementations include pointer verification and authority checks on MI objects. In addition, there are seven MI instructions that are unblocked and introduce an MI object into the user’s process (see Table 3-3 on page 3-41). These seven MI instructions perform DAC checks and are capable of generating audit records.

Table 3-3. Unblocked MI instructions which Introduce Objects into a Process

Unblocked MI Instruction	MI Object Introduced into Process
Activate Bound Program	Program
Activate Program	Program
Call External	Program
Create Independent Index	Index
Create Space	Space
Modify Space	Space
Transfer Control	Program

3.2.2.2.4. Special Privileged Instructions

There is a set of MI instructions that require the process to have a special authority in their associated user profile, group profile, or adopted authority. There is a 32-bit mask that is associated with each user profile and is used by the appropriate SLIC runtime routine to determine whether the user has the appropriate authority to execute the instruction. These instructions are blocked at translation time with the exception of Materialize Invocation Stack and Materialize Process Attributes. The implementations of these two instructions do not require special authority if a program is materializing its own invocation stack or attributes from the process in which it is running. They do require special authority if a program is attempting to materialize the attributes or invocation stack of another process. In addition to the appropriate pointer and authority checks, these MI instructions are capable of generating audit records.

The instructions that require privileged authority are as follows:

Table 3-4. List of Special Privileged Instructions

MI Instruction	Special Authority Required
Create Logical Unit Description	MI Privileged Instruction Authority
Create Network Description	MI Privileged Instruction Authority
Create Class of Service Description	MI Privileged Instruction Authority
Create Controller Description	MI Privileged Instruction Authority
Create Mode Description	MI Privileged Instruction Authority
Create User Profile	MI Privileged Instruction Authority
Destroy Authorization List	All Object[a]
Diagnose	MI Privileged Instruction Authority
Grant Authority	All Object[b]

Table 3-4. List of Special Privileged Instructions

MI Instruction	Special Authority Required
Grant-Like Authority	All Object[b]
Grant-Like Object Authority	All Object[b]
Initiate Process	MI Privileged Instruction Authority
Materialize Invocation Stack	Process Control[c]
Materialize Process Attributes	Process Control[c]
Materialize Process Identifiers	Process Control
Materialize Process User-IDs/Group-IDs	Process Control
Modify Authorization List	All Object[a]
Modify Machine Attributes	Modify Machine Attributes
Modify Process Attributes	Process Control[c]
Modify Process Identifiers	Process Control
Modify Resource Management Controls	MI Privileged Instruction Authority
Modify User Profile	MI Privileged Instruction Authority
Reclaim Objects	All Object
Request I/O	Service, Load, Dump[d]
Resume Process	Process Control
Send Signal	Process Control[c]
Suspend Object	Suspend Object Unrestricted[e]
Suspend Process	Process Control[c]
Terminate Instruction	Process Control[f]
Terminate Machine Processing	MI Privileged Instruction Authority
Terminate Process	Process Control[c]
Transfer Ownership	All Object[g]
Transfer Primary Group	All Object[b]

[a] Or object ownership or all private authorities

[b] Or object ownership or sufficient private authorities (depending on the specific operation)

[c] When operating on another process

[d] Special authorization required depends on the operation being performed

[e] Or Suspend Object Restricted and Object Control on the object

[f] When not the initiator of the process or the same process

g. Or object ownership

3.2.3. SLIC Functional Areas

This section describes the functional areas that comprise SLIC. These areas include the following:

- Tasks and task dispatching
- Queueing objects
- Storage management
- Process management
- Program activation and invocation
- Interprocess communication
- I/O management
- Exception management
- Event management
- MI translator

3.2.3.1. Tasks and Task Dispatching

The units of execution that may be run on a processor in the AS/400 are called tasks. A task defines the environment in which the work is to be executed and provides access to any storage that is required. In general, tasks are created as new work is initiated on the system and destroyed when the work has completed. Tasks are relatively short lived. Some tasks, however, act as servers for the operating system and are created at IPL time and destroyed at shutdown or re-IPL time. These tasks are long lived and provide the basic services for normal system operation but do not persist across IPLs. Tasks are non-persistent objects, that is, they are destroyed at system termination and recreated when needed during the next IPL. Therefore, no persistent object may contain the address of a task because the tasks address changes at the next IPL.

The content of a task reflects the current state of the unit of work to which it is associated when that work is not currently being performed on a processor. When a unit of work is being performed (i.e., the task has been dispatched to a processor and is currently executing), the current state of the unit of work is reflected in the content of the registers of the processor on which it is executing. For the following discussion of tasks and task dispatching, the term “task” refers to a non-executing unit of work. The phrase “currently executing task” refers to a unit of work that is loaded into and executing on a processor.

Tasks reside in E=R storage of the machine, which is accessible only in privileged instruction state as indicated by the MSR(PR) bit (MSR bit 49 = 1). This ensures that only operating system code can refer to the contents of a task. This also ensures that no page faults occur while trying to switch a task in or out of the processor, avoiding a recursion within the dispatcher. Tasks contain a copy of the contents of all general purpose registers as well as the contents of the special purpose registers used by the task at the time the task was switched out. These registers represent all the hardware information necessary to start or continue execution of the task. Floating point register contents are saved to a task only if a floating point instruction had been executed and are loaded from the task only when a floating point instruction is first executed after the task is switched in to the processor. This is done to improve performance.

When a task becomes eligible to execute, it is placed on the global dispatcher queue (also called the Task Dispatching Queue [TDQ]). The global dispatcher queue can be thought of as an array of linked lists of tasks (represented by a structure called a Task Dispatching Element [TDE]). The tasks on each linked list have certain characteristics in common, which dictate that they be grouped together for better performance. Tasks are placed on the global dispatcher queue by queueing operations (such as having a receive satisfied) or by task creation. The global dispatcher queue can be seen in Figure 3-12. “Global Dispatcher Queue” on page 3-44.

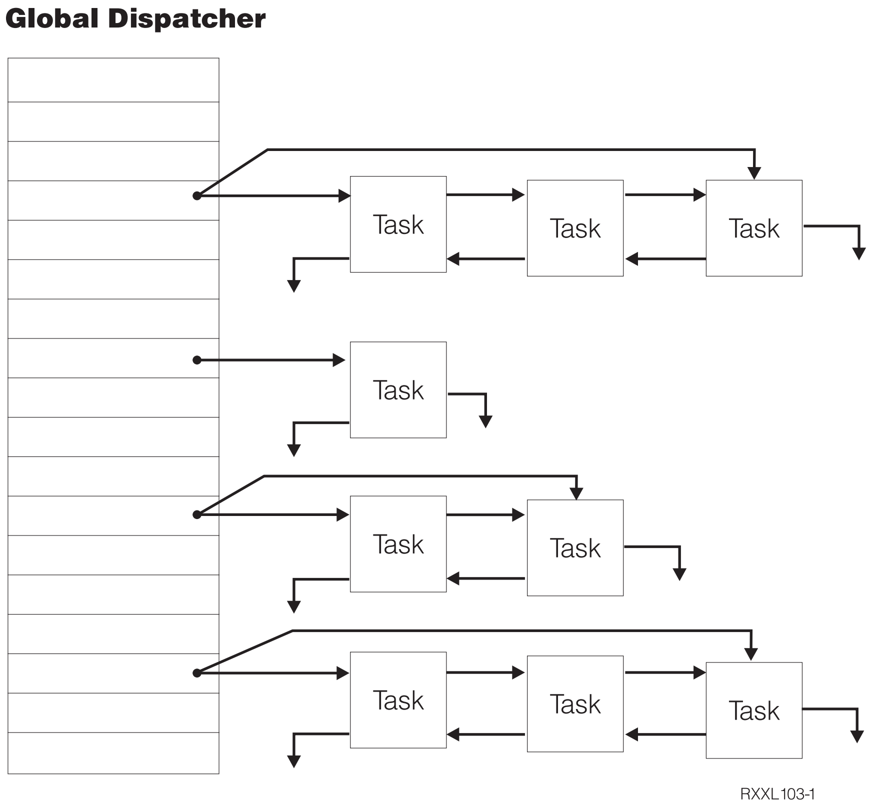


Figure 3-12. Global Dispatcher Queue

The selection of which task to execute next is based on the concepts of delay cost scheduling. The concept of delay cost scheduling is based on the premise that the importance of executing a task is proportional to the amount of time the task has been waiting to execute. The measurement of importance is the cost of the task and is valid only for the instant of time in which the cost was calculated. This is because the cost dynamically changes with time. The function of the dispatcher is to reduce the total cost of all tasks ready to execute. This requires that the dispatcher determine the highest cost task and ensure that it executes. After execution, the cost of the task is reduced.

Critical operating system tasks, however, require immediate, or near immediate, dispatch. To accommodate these tasks, a set of delay cost curves have high initial costs and a slope of zero. These are termed “flat curves” because the cost is constant over time. The dispatcher always selects the tasks that are on flat curves first, even if the cost of a task on a non-zero slope curve has exceeded the cost of the task on the flat curve. This is because the tasks on flat curves are assumed to be critical to the system operation and are “trusted” tasks.

The global dispatcher queue structure can be thought of as an array of linked lists. The tasks on each linked list are all on the same cost curve and thus all tasks on a given linked list increase in cost at the same rate. When a task becomes ready to run and is passed to the global dispatcher to be enqueued onto the dispatcher, the task is enqueued last on the link list, which is associated with the cost curve on which the task resides. In this way, the task at the head of the list is the oldest task on the linked list, and thus has the highest cost. The dispatcher then need only compare the costs of the tasks at the head of each linked list to determine which is the highest cost.

The dispatcher also takes into account cache affinity when selecting a task to be dispatched to a particular processor. Cache affinity is used to dispatch a task to the processor that is most likely to contain residual data in its cache and, therefore, the processor on which it will likely experience the best performance. The dispatcher assigns affinity to a task when it is dispatched to a processor. The task's affinity to the processor is initially strong, but starts to decay as soon as another task is dispatched to that processor.

There are two cases in which a task may be assigned to a curve other than the one it was created on during the course of execution. The first case is if the task violates the "contract" of the curve. Each curve has associated with it a set of limits on execution time and number of I/Os. Periodically, the execution time used and the number of I/Os performed are checked against the limits of the curve on which the task resides. If the task has exceeded either, then it is placed on the next curve down. This may be either a curve with a lower initial cost or a curve with a lower slope. The effect is that the task is less favored than it was before. When the task reaches an MI boundary, then it is returned to its original curve and given another chance. This is one of the mechanisms to prevent a task from "hogging" the system.

The other case in which a task can be placed on another curve is when it holds a resource that is required by a task which is on a curve with a higher slope. In this case, the task is placed on the higher slope curve until it relinquishes the resource, and then is returned to its original curve. This is done to allow the task on the higher slope curve to be serviced as quickly as possible without having to wait for relatively unimportant tasks to be serviced. The act of placing a task on a higher slope curve is termed "bumping" the task. If a task is holding multiple resources, it is possible that it could be bumped multiple times. A count is kept within the task which indicates how many times the task has been bumped. It is incremented when the task has its curve bumped and is decremented when it relinquishes any resource which has caused a bump. The task l remains on the highest slope curve that it acquired during multiple bumps until the count returns to zero.

Every OS/400 active job, see section 4.1.3.3 "Job Management" on page 4-23, runs as a process, see section 3.2.3.4 "Process Management" on page 3-61, and is represented by a SLIC task. However, there are always more SLIC tasks than OS/400 processes. The SLIC tasks that are not in one-to-one correspondence with OS/400 processes enable asynchronous activity within the SLIC layer. These SLIC tasks with no corresponding OS/400 process are referred to as asynchronous SLIC tasks. They are similar to daemons in other systems. The asynchronous SLIC tasks are defined by the SLIC components that require the asynchronous function. For example, the SLIC Load/Dump component creates asynchronous SLIC tasks to save or restore MI objects; see section 4.3.2 "Backup and Recovery" on page 4-90 for more about this topic. Another example is SLIC I/O Management, discussed in section 3.2.3.7 "SLIC I/O Management" on page 3-79. A third example is the asynchronous SLIC task used by process management to complete termination of a process and then clean up for the terminated process.

The asynchronous SLIC task is created and queued for dispatching by the SLIC process management component.[12] After the asynchronous SLIC task's TDE has been dispatched, it is typically put on a wait queue to sleep while waiting for the arrival of a message sent by the SLIC task requiring its function.

These asynchronous SLIC tasks are not visible outside of the SLIC layer; they have no interfaces to OS/400 or to the end user; no MI instructions can refer to them.

When an asynchronous SLIC task is no longer needed (e.g., when a device is varied off, as discussed in the following section), the task destroys itself by issuing a process management macro.

3.2.3.2. Queueing Objects

The Queueing component is responsible for providing the low level synchronization of tasks required by SLIC. The following sections describe several types of "wait objects." A wait object is an object on which a

[12] The SLIC task is initiated by a SLIC component issuing the TSKCRT macro. This macro invokes the SLIC routine CFCTASK, which obtains a SLIC storage segment, using the CRTSID macro. The storage for the Task Control Block (which includes the TDE) is allocated from the main storage page frame (V=R address). The Task Control Block is initialized along with the first ICB. The address of the TDE is then enqueued on the TDQ for dispatching. Unlike an MI process, the SLIC task has only one storage segment that contains the Task Control Block and the invocation work area. The Task Control Block of the SLIC task contains a subset of the information used in the MI process.

task may “block” when performing a receive type of operation. Blocking the running task causes the task dispatcher to select the next highest priority task and make it the running task. The blocked task’s state is saved and it is chained as a waiter on the object. A task remains blocked on an object until another task performs a send type of operation on the wait object that satisfies the receive operation of the blocked task.

3.2.3.2.1. SLIC Message, Queue, and Pool Objects

A running task sends information to another task by creating a formatted data structure called a Message (also called a Send/Receive Message [SRM]). A SLIC queue (known as a Send/Receive Queue) is a data structure that is used as a “mailbox” for messages sent from one task to another. See Figure 3-13. “Task Waiting Queue” on page 3-46 for a conceptual view. If a task performs a receive operation from a queue that has no messages, or if no message satisfies the key value specified by the receive operation, the running task blocks on that queue. The blocked task waits on the queue until another task sends a message to the queue that satisfies the original receive from the blocked task. The waiter list for the queue is examined whenever a message is sent to it to determine if any tasks are waiting for that message. If there are, the waiting task or tasks are put on the task dispatcher queue (TDQ) in priority sequence.

A SLIC queue can be in ascending, descending, or unsorted order. The task dispatching queue is an example of a SLIC queue in ascending order. An “equal order” can also be specified, which is used to determine how to handle messages with equal (duplicate) keys: “first of equals” puts a message before its equals, “last of equals” puts it after. An unsorted queue with “first of equals” acts like a last-in first-out (LIFO) queue; with “last of equals,” an unsorted queue behaves like a first-in first-out (FIFO) queue.

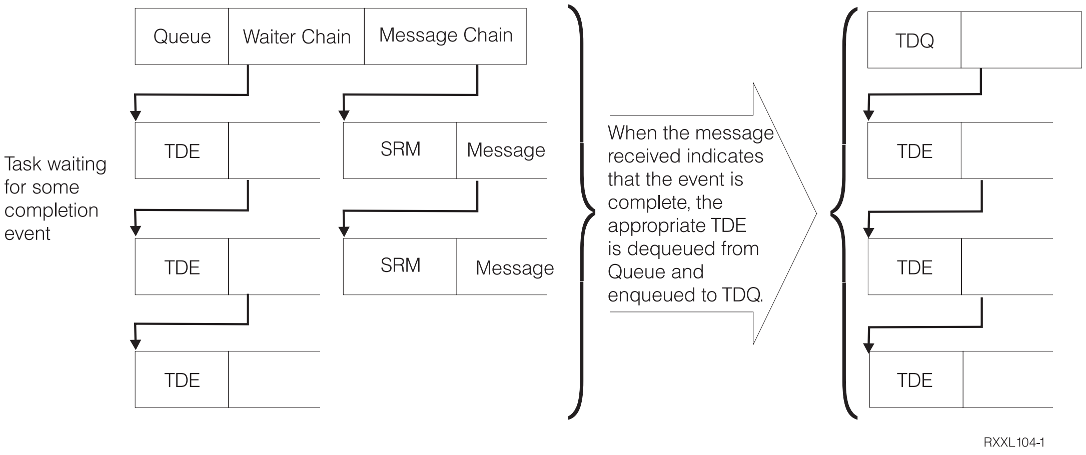

Queue
Waiter Chain
Message Chain
Task waiting for some completion event
TDE
TDE
TDE
SRM
Message
SRM
Message
When the message received indicates that the event is complete, the appropriate TDE is dequeued from Queue and enqueued to TDQ.
TDQ
TDE
TDE
TDE
RXXL104-1

Figure 3-13. Task Waiting Queue

A SLIC Pool is a “cache” of some number of internal SLIC objects such as messages. The upper and lower limits of the number of objects that can be in the pool can be set by the pool user. If an object is available in the pool when requested, it is returned to the user. If the object is not available, a new object is created. If the number of objects already used from the pool is at the maximum, the calling task blocks until an object is returned to the pool.

3.2.3.2.2. Semaphore-like Objects

A SLIC Counter (also called a Send/Receive Counter) is a semaphore-like object that is used to synchronize or serialize tasks without exchanging messages. A Counter contains a binary counter, a limit value, and a waiter list. When an executing task sends to a counter, the binary counter is incremented by one and compared to the limit value. If the binary counter equals or exceeds the limit value, one task on the waiter list is removed from the list and put on the task dispatcher queue in priority sequence, and the count is decremented by the value of the limit. If an executing task performs a receive operation from a counter whose binary counter is less than the limit, the task is blocked and placed on the counter waiter list.

There are two special forms of SLIC counters:

1. A SLIC State Counter is a counter that does not have a limit field. The compare value and compare type are specified on a receive operation. If the comparison is not true, the task blocks and is placed on the waiter list. A task can decrement the counter value as well as increment it.
2. A SLIC Single Task Blocker is used to block only one task. Multiple tasks may unblock this object, but the number of unblocks on the object is not tracked. If the object is found unblocked when a block is attempted, the task will not block. In this case, the unblocked signal will be cleared such that subsequent attempts to block will cause the calling task to block.

3.2.3.2.3. Mutual Exclusion Objects

A SLIC Gate is an object that can be used to synchronize access to sections of code or shared data. A gate can be locked exclusive (writer lock) or locked shared multiple times (multiple readers). When a requested lock type conflicts with the current lock status of a gate, the requesting task is blocked until the gate is unlocked. When a lock is released, the blocked lock requests are granted to the waiters in priority order. The holder of an exclusive lock may have its priority bumped to the level of the waiter to expedite the release of the lock. Gates also support a “try lock” operation, which allows a gate user to attempt to obtain a lock. If the requested lock conflicts with an existing lock, the task does not block.

There are two special forms of a SLIC gate:

1. A Try Gate that supports a “try unlock” operation, which allows the holder to determine if a “try lock” occurred while the lock was held.
2. A MultiTry Gate works just like a Try Gate, but a task may lock the gate multiple times. The gate must be unlocked once for each time it is locked.

3.2.3.3. Storage Management

The Storage Management (SM) component is responsible for managing three system resources: virtual storage, auxiliary storage (i.e., direct access storage device [DASD]), and main storage (i.e., dynamic random access memory [DRAM]). It can be invoked directly by other SLIC components via a set of interfaces, or it can be invoked indirectly as a result of a page fault.

This section describes storage management as it is designed on the AS/400. It first describes the three system resources managed by SM: auxiliary storage, main storage, and virtual storage. This section then describes auxiliary storage management (ASM), which is the part of storage management that manages the allocation of virtual storage and auxiliary storage, and main storage management (MSM), which is the part of storage management that manages DRAM. This section also describes the I/O functionality of SM. Finally, the actions performed by SM during start-up and shutdown of the system are explained.

3.2.3.3.1. Definitions

The following definitions are used throughout the remainder of this section.

auxiliary storage All addressable disk storage.

main storage All addressable storage where programs are run. Synonymous with memory.

pages A unit of measure for virtual address space, defined as 4096 bytes.

segments A unit of measure for virtual address space. *Big segments* are 16 MB (4096 pages) of virtual address space, and *little segments* are 64 KB (16 pages) of virtual address space.

frames The unit of measure for main storage, defined as 4096 bytes. These are also referred to as *pages* of main storage.

sectors The unit of measure for auxiliary storage. Sectors are 520 bytes in length. However, the first 8 bytes is the *sector header*, which is used by storage management. Consequently, each sector has 512 bytes available for data. Eight sectors together make up one page, and the 8 sector headers are combined into a logical 64-byte *page header* (see section 3.2.3.3.6 “Page Header” on page 3-56).

extents Contiguous grouping of pages. Extents have any of the following sizes: 1, 2, 3,4, 5, 6, 7, 8 and in power-of-two number of pages up to 16 MB (i.e., 16 pages, 32 pages, up to 4096 pages).

3.2.3.3.2. Virtual Storage and Single Level Store

Single level store is a single, device-independent, addressing mechanism that provides transparent storage addressing, making both main storage (memory) and auxiliary storage (DASD) appear to be continuous. The system’s directory (contexts) contains virtual addresses rather than real disk locations. To access an object, the MI user simply references it by name or pointer. This causes the machine to automatically resolve the address of the object. The underlying virtual addressing mechanism ensures that the object gets loaded from auxiliary storage into main storage.

AS/400 virtual addressing is completely independent of an object’s physical location, and the type, capacity, and number of disk units on the system. An object may be stored by distributing it across several disk locations on several disk units, yet its contents have contiguous virtual addresses.

The AS/400 system supports a single system-wide virtual storage space. This virtual storage space is addressed with an 8-byte address, which yields a total addressable virtual storage of 16,777,216 terabytes (address range: 0 through 2^{64}-1).

SM logically divides this 16,777,216-terabyte virtual storage space into non-overlapping areas, called *segments*. Because the portions of AS/400 that manage big and little segments “know” which they are dealing with and because the same concepts apply to both, the remainder of this section refers to both simply as *segments* unless the distinction is relevant.

Storage management has a set of interfaces that allow SLIC components to “allocate” segments from a free pool of unallocated segments and return segments to this free pool. Each segment is referred to by its 8-byte virtual address, which is returned by the interface when a segment is “allocated.”

SM breaks the 16,777,216-terabyte virtual address space into the following ranges as illustrated in Figure 3-14. “AS/400 Virtual Address Space” on page 3-52:

Permanent Preassigned Segments: A range of virtual address space reserved for SLIC use.

Permanent Storage: A range of the virtual address space reserved for “permanent” storage. Any segment allocated from this range remains allocated until the segment is explicitly freed (via an SM interface). Some segments are “reserved” and, therefore, not available to be allocated.

Not Used: Ranges of virtual address space that are not currently used or assigned by SM.

Real (main) Storage: A range of the virtual address space (called E=R addresses) is reserved for main storage (i.e., DRAM) (see section 3.2.3.3.7 “Main Storage” on page 3-56). The ASM interfaces do not allocate virtual space from this range of addresses, and the MI instructions do not provide an interface to this range of virtual addresses. However, Resident Storage Management (RSM) interfaces exist that allow a user to allocate a *resident segment*. A resident segment is one or more pages of main storage that are allocated and assigned a virtual address from the temporary address range, but have no backing storage on DASD. This gives the storage management user the capability of allocating more than one page of real memory but not having to manage the individual pages (i.e., they can refer to the entire range with a virtual address). This range of addresses can also be accessed by RISC instructions. Depending on the machine model, the amount of actual DRAM ranges from 64 megabytes to 20 gigabytes. Referring to an address that does not correspond to an actual DRAM location causes an exception.

I/O Addresses: A range of virtual address space reserved for use by IOPs for performing I/O operations.

Temporary Storage: A range of the address space reserved for “temporary” storage. Any segment allocated in this range that is not explicitly freed (using an SM interface) is implicitly freed by SM at shutdown or the next IPL. Some segments are “reserved” and, therefore, not available to be allocated.

Access Group Storage: A range of virtual address space reserved for a special form of “temporary” storage known as “access-group” storage. Any space allocated in this range that is not explicitly freed (using an SM interface) is implicitly freed by SM at shutdown or the next IPL. Access-group storage (along with the associated access-group MI objects) is used to enhance the performance of moving the temporary segments associated with a given job into and out of main storage. From a security viewpoint, access group storage is handled identically to temporary storage.

Microcode Segments: A range of virtual address space reserved for all LIC code.

3.2.3.3.2.1. Segment Identifier

SM assigns each big segment a unique *segment identifier (SID)*, which is the high-order five bytes of the 8-byte virtual address; the low-order three bytes of the 8-byte virtual address are used as the offset into the big segment, and is called the *segment offset*. Similarly, the SID associated with each little segment is the high-order six bytes of the 8-byte virtual address and the segment offset is the low-order two bytes.

3.2.3.3.2.2. IPL Wrap

When the total number of available permanent or temporary SIDs are exhausted, SLIC code forces all permanent objects to DASD, raises an exception indicating that the SID limit has been reached, and causes the system to go down ungracefully. For the system to be usable again, it must first be brought out of the evaluated configuration, and the Auxiliary Storage Initialization Dedicated Service Tool (DST) procedure must be run. Following this DST procedure, the system must be re-installed (scratch install) according to the procedures documented in *Security - Enabling for C2* [10].

3.2.3.3.2.3. Reserved Segments

Some segments are reserved for use by storage management and by other SLIC components. For example, the SLIC code, the machine context, the storage management directories (see section 3.2.3.3.8.1 “Storage Management Data Structures” on page 3-57) and the main storage dump space (see section 3.2.3.3.11.2 “Storage Management Shutdown” on page 3-61) all reside in reserved permanent segments.

3.2.3.3.2.4. Reusable Segments

Reusable segments are temporary segments that can be used multiple times within an IPL. This feature was added to reduce segment consumption. The deallocator of a temporary segment may specify that it is reusable, and SM maintains a pool of free reusable segments. A segment allocator may specify that it will use a reusable segment. By default, SM zeros reusable segments after being re-allocated and before control is returned to the requestor of the segment.

3.2.3.3.2.5. Holey Segments

Temporary segments are also allowed to be "holey" segments. When a segment is created, 16 MB of addressability is allocated (for a big segment), but anywhere from 4 KB up to 16 MB may actually be allocated at any one time. For a non-holey segment, whatever part is currently allocated starts at the lowest address for the segment (offset 000000), and is contiguous up to the highest address currently allocated. This does not mean that the extents allocated on disk are all contiguous, but that the range of addressability for this segment that is currently allocated must be contiguous. For holey segments, however, the allocated part of the segment does not have to all be contiguous, i.e., there can be "holes" in the allocated part of the segment.

One use of holey segments is for stack segments. In this case, the first page of the segment is allocated, and then when stack frames are created while a task is running, the stack frames are allocated from the highest addresses in the segment backwards. A stack segment has only one "hole" in its allocated storage. This is not a limitation of holey segments. A 16 MB temporary segment could have every other page of its 4096 pages allocated and would then have 2047 "holes."

3.2.3.3.2.6. Access Group MI Object Overview

Since access group storage is closely aligned with access group MI objects, this section presents an overview of access group MI objects and describes how they are supported by Storage Management.

An access group MI object is used to collect a set of other MI objects together on auxiliary storage so that they can be transferred to and from main storage with a fewer number of I/O operations. When an access group is created, the DASD sectors for its segments are allocated from a contiguous block of auxiliary storage. An MI object may then be created "into" the access group by specifying a pointer to the access group in the object's create template.[13] The segments that compose these MI objects are assigned their own unique SIDs from the access group virtual storage, but their auxiliary storage space is suballocated from within the access group auxiliary storage space.

An access group MI object consists of two segments. The first segment, called the table of contents segment (or TOC segment), contains the encapsulated program architecture (EPA) header and the object-specific header for the access group MI object, and the access group table of contents. The second segment, called the access group storage segment, is used to contain the MI objects in the access group. For each page in the access group storage segment, there is an 8-byte entry in the TOC segment. Each TOC entry, by virtue of its location in the TOC segment, refers to a unique data page in the access group storage segment. The TOC entry contains the virtual address of the data currently residing in its associated data page.

Access group MI objects are created, materialized, and destroyed via MI instructions provided by the Resource Management component. Auxiliary Storage Management (ASM) provides interfaces to create, extend, truncate, and destroy segments in an access group. These interfaces are called by the appropriate MI object management MI instructions.

Main Storage Management (MSM) provides interfaces to transfer all MI objects in an access group to and from main storage as a single unit. Individual objects within an access group may also be paged in and out separately.

[13] Storage Management has interfaces to create segments into an access group.

Access Group MI objects and MI objects created into access groups are created from access group virtual storage. Since this type of virtual storage is temporary, neither exist over IPLs.

3.2.3.3.2.7. Segment Header

Each segment has a 32-byte *segment header*, which includes the following fields:

- **The Size of the Segment** - in 512 byte units up to a maximum of 32768 units.
- **Hardware Storage Protection Bits** - Four possible values:
  - Read/write in either user or system state
  - Read only in user state
  - No access to segment in user state
  - Read only access in either user or system state
- **Multiple Hardware Storage Protection Bit** - indicates whether this segment contains multiple hardware storage protection values.
- **Hardware Storage Protect at all Security Levels Bit** - indicates whether this segment's hardware storage protection bits are enforced at all system security levels. In the evaluated configuration, the hardware storage protection bits are used and checked.
- **Base Segment Address** - if the segment is one of a set (for example, an MI object), this pointer contains the address of the base segment (the segment that contains EPA header).
- **Associated Space Address** - This is used to contain the address of the associated space. It is not initialized by SM but by the object manager.
- **Segment Domain -** System or User. Set by object manager, not by SM.

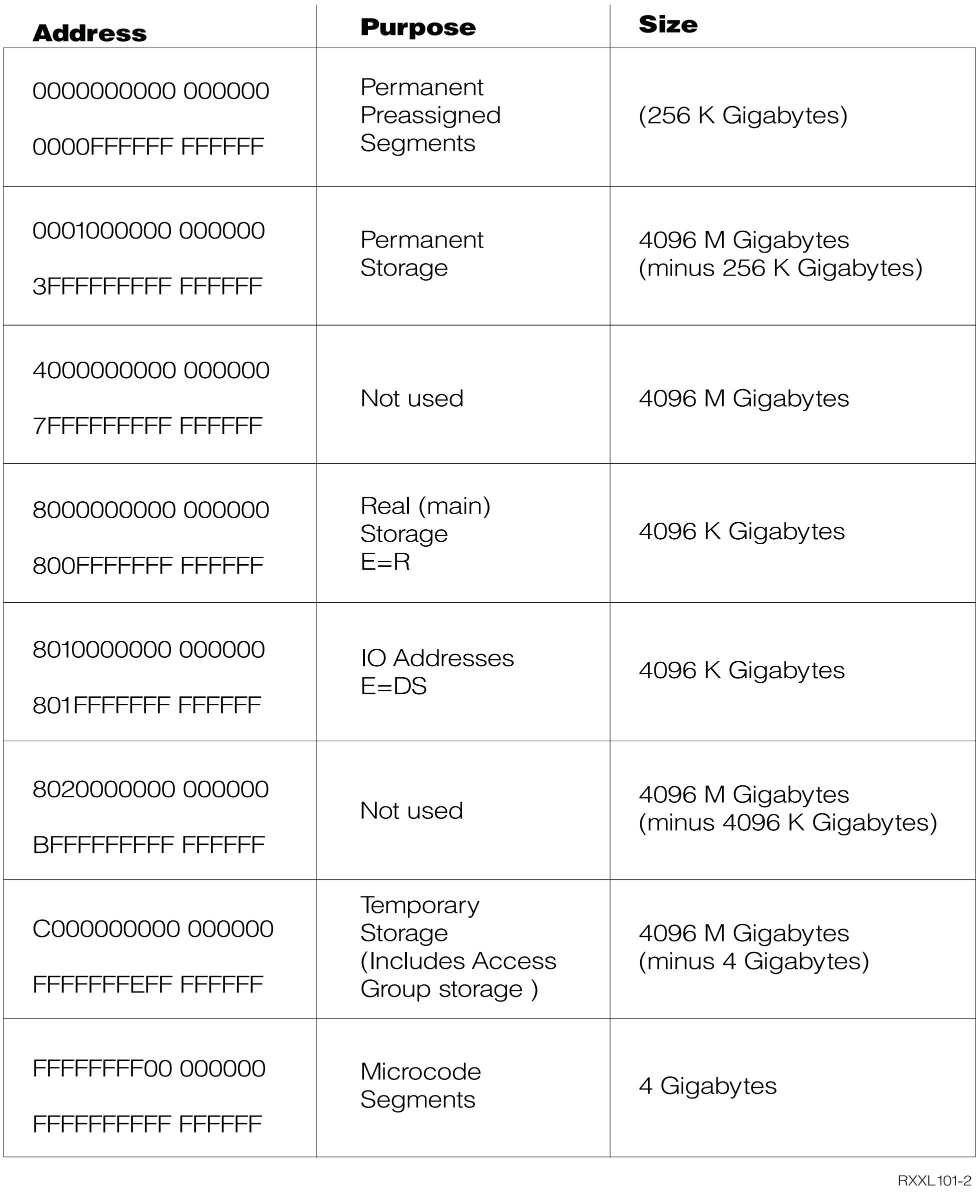
3.2.3.3.3. Addresses and Address Translation

When an object is created, the SLIC storage management function assigns an 8-byte effective (virtual) address. This address remains unchanged for the period the object exists on the AS/400. See section 3.2.3.3 “Storage Management” on page 3-47 for details. A real address exactly corresponds to a specific byte of main memory. A virtual address is subject to being remapped (transparently to the user) to any location in the memory of the system, either in main storage or in secondary storage.

Each virtual address requires translation to a real address so that the correct location in storage can be accessed.

Addresses used by the AS/400 at the RISC level are 64 bits in length. These 8-byte (64-bit) addresses are supported directly by the hardware. They are mapped as follows:

- Address of 64-KB segment = 48-bit segment ID + 16-bit offset
- Address of 16-MB segment = 40-bit segment ID + 24-bit offset



Address	Purpose	Size
0000000000 000000 0000FFFFFF FFFFFF	Permanent Preassigned Segments	(256 K Gigabytes)
0001000000 000000 3FFFFFFFFF FFFFFF	Permanent Storage	4096 M Gigabytes (minus 256 K Gigabytes)
4000000000 000000 7FFFFFFFFF FFFFFF	Not used	4096 M Gigabytes
8000000000 000000 800FFFFFFF FFFFFF	Real (main) Storage E=R	4096 K Gigabytes
8010000000 000000 801FFFFFFF FFFFFF	IO Addresses E=DS	4096 K Gigabytes
8020000000 000000 BFFFFFFFFF FFFFFF	Not used	4096 M Gigabytes (minus 4096 K Gigabytes)
C000000000 000000 FFFFFFFEFF FFFFFF	Temporary Storage (Includes Access Group storage)	4096 M Gigabytes (minus 4 Gigabytes)
FFFFFFFF00 000000 FFFFFFFFFF FFFFFF	Microcode Segments	4 Gigabytes

RXXL101-2

Figure 3-14. AS/400 Virtual Address Space

The method used to translate between a virtual address and a real address depends on the value of the virtual address.

1. A real address (also referred to as a Effective=Real or E=R address). If the high-order 12 bits of the 64-bit address are equal to X'800', an effective=real (E=R) address is assumed, and the main storage address is formed by ANDing off the high order 12 bits.
2. A virtual address. This is any 8-byte address not in the range of valid E=R addresses. A virtual address is translated by the hardware hashing of the SID and page ID (PID) to select an entry from the hardware page table (HPT), which in turn specifies a location in main or secondary storage. The

HPT is a structure whose format and content are described in section 3.1.1.1.1 “Hardware Address Translation” on page 3-8. Since the structure of the HPT may prevent it from holding all valid virtual-to-real translations, SLIC has implemented a structure, called the software page table (SPT) that does contain all valid translations, and additional information on very page frame of main storage. When the hardware portion of the translation fails, a data store interrupt is signalled. SLIC SM receives control and uses the SPT to determine the address translation, reads the virtual page from auxiliary storage if necessary, and updates the HPT with the translation information; then the hardware retries the original faulting instruction. The following paragraphs discuss the SLIC portion of this process in detail.

The SPT is composed of one 64-byte entry called a page descriptor (PD) for each 4096-byte page of main storage on the system. The virtual to real translation is implied by the position in the SPT in which an entry exists. If the second entry in the SPT contains a virtual address of X’1234567890123xxx’, then that virtual address translates to a real address of X’0000000001xxx’ (i.e., the second page of main storage). An SPT entry includes the following fields in addition to the implicit real page address:

- A field indicating that the page is not eligible to be paged out; it is “pinned” in storage.
- A pointer to the next software page table entry in the hash chain (X’FFFF’ indicates the end of the hash chain)
- The segment and page identifiers of the page represented by this entry
- A set of bits indicating whether this page:
  - is valid (disk copy has been read into main storage and main storage now contains the most recent values for the page)
  - has been referred to recently
  - has been changed (in main storage)

  These indicators are used to determine when a page should be saved to, or retrieved from, secondary storage:
- A pair of bits indicating the processor storage protection state of the page
- The unit number and relative sector number of the disk device where the page is stored

The page tables themselves are stored in E=R memory and are *always resident.* The address resolution procedure is illustrated by Figure 3-15. “SLIC Address Translation” on page 3-54, which is numbered to follow the steps described below.

To resolve a virtual address (1) in the SPT, the segment identifier (SID) and page identifier (PID) of the virtual address are hashed into a 32-bit hash table entry offset. The real address of the selected hash table entry is then obtained by adding this 32-bit hash table (2) offset value to the starting address of the SPT (3). The page descriptor (PD) stored at this location in the SPT is retrieved, and the segment and page identifiers from the PD are compared with the SID and PID of the virtual address to be translated. If they differ, the next PD along the hash chain is retrieved, and the same check is made. Details of a PD are illustrated in Figure 3-16. “Page Descriptor (PD)” on page 3-55. Once this check succeeds, the valid bit is checked. If the page is valid, then the real address (4) is formed by using the high-order 40 bits of the offset of the current PD from the start of the PD as the page frame identifier and concatenating the twelve bits of the byte identifier as the low-order portion of the address. If the page is not valid, it must be retrieved from the disk unit and sector specified in the PD. See section 3.2.3.3 “Storage Management” on page 3-47 for details about how this is handled.

The real address is used to read the cache and the cache directory in parallel. If the real address for the 128-byte cache line in the cache directory matches the real address to be accessed, the cache data is sent to the processor. Otherwise, the data has to be read from memory, written into the cache, and sent to the processor.

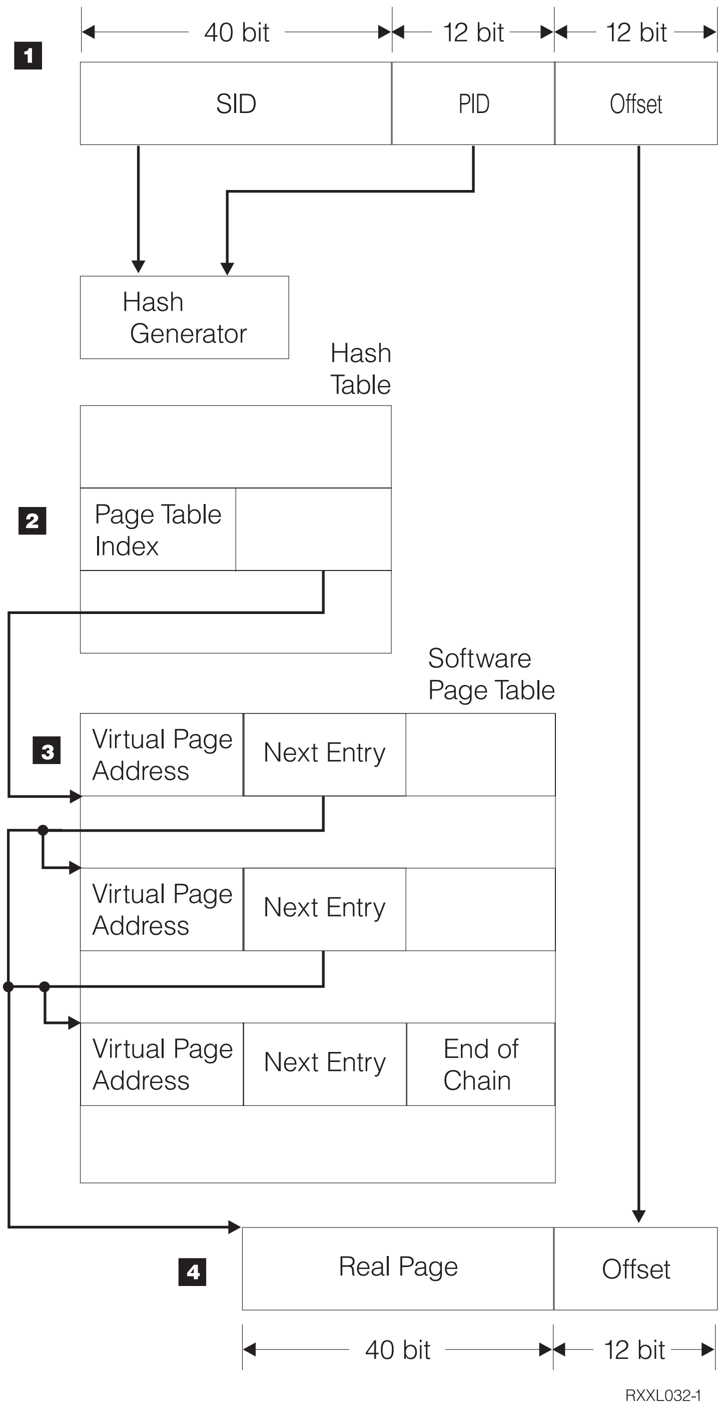


Figure 3-15. SLIC Address Translation

Because the cache is a store-through (i.e., cache writes are also sent to memory), the cache entry written does not have to be purged; only the cache directory is purged.

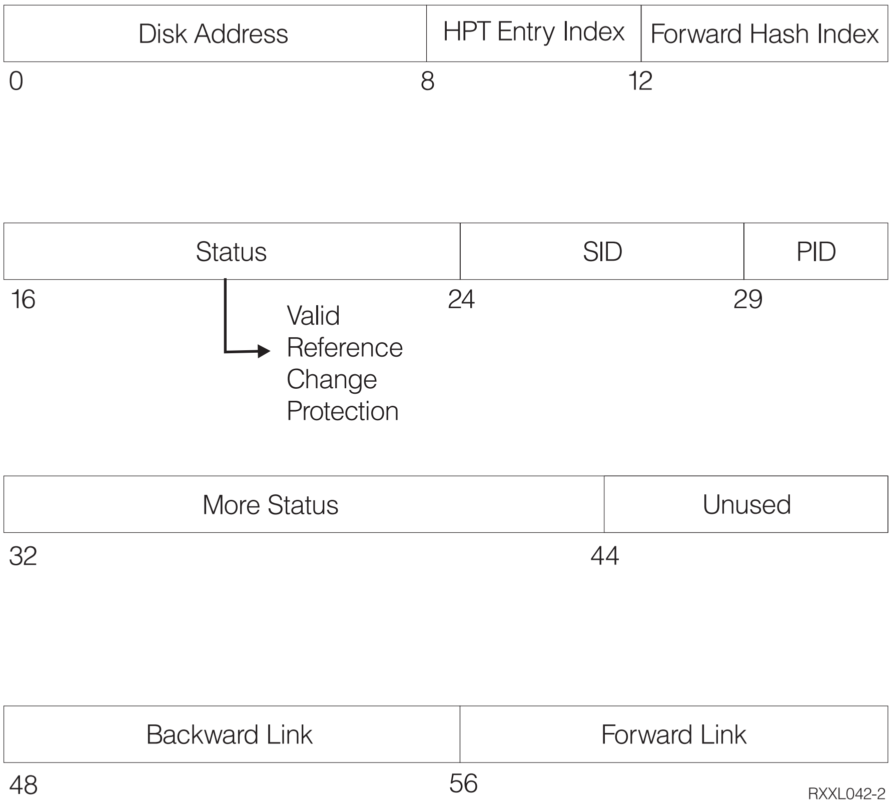


Figure 3-16. Page Descriptor (PD)

3.2.3.3.4. Storage Protection

The storage protection bits discussed in the SPT section above are used as described in section 3.1.1.1.3 “Hardware Storage Protection” on page 3-10 to determine protection if the Machine State Register (MSR) is set to user state. If the MSR is set to system state, the protection state is not checked.

3.2.3.3.5. Auxiliary Storage

Storage Management (SM) is the manager for auxiliary storage, as described in section 3.2.3.3.8 “Auxiliary Storage Management” on page 3-57. SM divides DASD into 4160-byte *pages*.[14] On each page, 64 bytes are reserved for the *page header* as described below.

[14] Each page consists of eight 520-byte sectors. Each sector can contain 512 bytes of data and 8 bytes of header. Therefore, a 4,160 byte auxiliary storage page can contain 4,096 bytes of data and 64 bytes of header.

3.2.3.3.6. Page Header

The page header is a 64-byte header for each page on disk. It contains the following fields:

- **Virtual Address**: This is the virtual address of the segment that this page represents. This information is used to rebuild the permanent directory and free space maps in the event of a system crash. This is also used to provide the zero on first reference function.
- **Tags**: Locations of all tagged pointers within the page. This information is used to restore tag bits when the page is loaded into main storage.
- **Logically Bad Flag**: Physically good page that represents a page that had a read error. The data is corrupted. SM uses this flag to return a read error on this page. This page must be cleared and written to reset this flag. Object managers typically mark an object as “damaged” when receiving a read error.
- **IPL Number**: The IPL number when this page was last written to DASD (only used for debug).
- **IO Count**: This is the system IO count[15] for which this page was last written (only used for debug).
- **Storage Protection Information**: Storage protection value for this page.
- **Extent Size Code**: An indication of the size of the extent that this page is part of.

3.2.3.3.7. Main Storage

As previously mentioned, the amount of main storage (e.g., DRAM) can vary from 64 megabytes to 20 gigabytes. Part of this storage is reserved for memory-resident data structures (e.g., I/O buffers) and for the SLIC nucleus. The remainder of main storage is available for paging.

SM is the manager of the main storage that is available for paging. At IPL, SM uses information supplied in the Main Storage Vital Product Data (VPD) object to divide available main storage into frames and to build the page tables that provide the ability to translate a virtual address to its equivalent real address in main storage.

SM also partitions main storage into one or more *storage pools*. For each storage pool, SM builds the four *lists* described below:

- **Unavailable List**: A list of frames for that storage pool that are not available for paging.
- **Unused List**: A list of frames for that storage pool that are available for paging, but have no virtual address assigned to them.
- **Available List**: A list of frames for that storage pool that are available for paging and have a virtual address currently assigned to them.
- **Change List**: The list of changed frames waiting to be written to auxiliary storage. SM writes changed frames back to auxiliary storage when the machine is not busy or when there are no steal candidates in the available list, and then transfers the frames to the available list.

[15] Taken from the atomic system IO count.

3.2.3.3.8. Auxiliary Storage Management

ASM is responsible for the following:

- Allocating virtual storage to SLIC routines
- Managing the free space on DASD devices
- Maintaining data structures that map 8-byte virtual addresses to disk locations

It is invoked from other SLIC routines through a provided interface.

ASM allocates virtual storage in segments. When a SLIC routine requests a segment, it specifies whether it wants a little segment or a big segment (64 KB or 16 MB), the initial DASD allocation required for the segment, whether the request is for a permanent or temporary segment, and whether it is an access group segment. These segments may be used for objects or for SLIC internal use. When the segment is allocated, the unique 5-byte ID (SID)[16] is returned to the calling SLIC routine.

3.2.3.3.8.1. Storage Management Data Structures

ASM maintains a set of data structures that are used to keep track of the contents of disk space. These are as follows:

Free Space Bit Maps: This data structure is a multilevel bit map that tracks the disk location of each extent of free auxiliary storage.

Permanent Directory: This data structure contains an entry with the disk locations of extents that have been allocated to permanent segments. It also contains the hardware storage protection associated with that segment.

Temporary Directory: This data structure contains an entry with the disk locations of extents that have been allocated to temporary segments. It also contains the hardware storage protection associated with that segment. This directory also contains entries for access groups also.

Static Directory: This data structure is a list of extents that have been allocated to pre-assigned, permanent segments. It is used to locate segments that are essential for machine execution when the permanent and temporary directories are unusable.

Lookaside Directory: This data structure is a list of recently referenced extents from the permanent and temporary directories. It is essentially a “cache” for the permanent and temporary directories. Its purpose is to avoid the relatively lengthy index operations on these directories.

Access Group Table of Contents:This data structure contains an entry with the disk location of each extent of free and allocated auxiliary storage assigned to access groups.

3.2.3.3.9. Main Storage Management

Main Storage Management (MSM) is responsible for the transfer of data between main storage and DASD (including management of the *tag bit*), and for the management of the main storage resource (i.e., frames). It is invoked by other SLIC routines implicitly through page faults and explicitly through interfaces. MSM is a

[16] The segment identifier (SID) is 5-bytes. When the SID appears as part of a segment address, it occupies 8-bytes.

cooperative effort of the AS/400 hardware and the storage management (SM) component of SLIC. This section describes the support provided by SLIC.

The MSM portion of SLIC Storage Management is responsible for allocating frames of main storage and for attempting to copy the data associated with the virtual storage pages from DASD into main storage and back.

It also saves and restores any tag bits, sets the hardware storage protection bits in the page table entry for each frame as appropriate, and zeros out the frame in main storage if this is the first reference to the virtual storage page (see section 3.2.3.3.9.4 “Zero on First Reference” on page 3-60).

The following sections describe the page fault processing performed by SM, the interfaces provided by SM, and management of the tag bits by SM.

3.2.3.3.9.1. Page Faults and Page Fault Processing

When a RISC instruction operand specifies a virtual address, the hardware attempts to translate the virtual address into a real address, as described in section 3.1.1.1.1 “Hardware Address Translation” on page 3-8. If this translation fails, a *data store interrupt*, commonly referred to as a *page fault*, is signalled to SLIC.

When signalling this interrupt, the hardware provides via registers: (1) the virtual address that caused the interrupt, and (2) the address of the instruction that was executing at the time of the interrupt.

If the virtual page is already resident in main storage, then MSM simply ensures the hardware page table (HPT) is updated with the required information to allow the hardware to successfully translate the virtual address into a real address. MSM then causes the faulting instruction to be retried.

If the virtual page is not already resident in main storage, then MSM attempts to bring the virtual page to which the faulting address refers into main storage (making it “resident and valid”) and causes the faulting instruction to be retried.

If the virtual page cannot be brought into main storage, then MSM throws one of the addressing exceptions described below (which explain why the page cannot be made “resident and valid”).

To perform this job, SM searches the appropriate SM directories (see section 3.2.3.3.8.1 “Storage Management Data Structures” on page 3-57) to determine whether or not the segment containing the page currently exists.

- If the segment does not exist, a “segment does not exist” exception is thrown, and page fault processing is terminated.
- If the segment does exist, SM determines whether the current size of the segment includes the faulting page.
    - If the current size does not include the faulting page, a “page not allocated” exception is thrown, and page fault processing is terminated. The faulting task can extend the segment to allocate the appropriate page.
    - If the current size includes the faulting page, the directory entry for the virtual page will contain sufficient information to determine the disk location (disk unit number and sector number) of the page, and page fault processing continues.
- MSM allocates a main storage frame from the correct storage pool. (Section 3.2.3.3.9.2 “Frame Steal” on page 3-59 describes the frame stealing algorithm in more detail.)

- MSM then updates the software page table (SPT) entry for the allocated frame so that the virtual page is “resident and invalid” while the read from DASD is performed. While the page is “resident and invalid,” any other tasks that incur a page fault exception on the same page are placed in a “I/O pending” state so the task waits for the I/O to complete. This avoids having the same virtual page read into more than one main storage frame.
- MSM then initiates the read from disk into the main storage frame.
  - If the read is unsuccessful, a “page read error” exception is thrown.
  - If the read is successful, MSM does the following:
    - Compares the sector header information with information in the directory to ensure that the data on disk represents the page being requested. If it does not, MSM clears the main storage frame.
    - Sets the hardware protection bits in the page table entry appropriately.
    - Restores any tag bits.
    - Makes the virtual page “resident and valid” by updating the SPT and HPT entries appropriately. Finally, the faulting Amazon instruction is retried.

3.2.3.3.9.2. Frame Steal

Each storage pool has two lists that participate in the frame steal process; an available list and a change list. The frame steal algorithm always searches the entire available list looking for a steal candidate. When this list is exhausted, the algorithm causes the associated task to enter the *page frame wait* state. The handling of this condition is described in section 3.2.3.3.9.3 “Page Out Task” on page 3-59.

The following frame attributes, which are contained in the frame’s SPT and HPT entries, are examined when the frame is being considered as a steal candidate:

- **Referenced Bit**: Indicates whether the frame has been referenced since it was read into main storage or was last examined as a steal candidate. This bit is set by RISC instructions that refer to the frame. MSM clears the bit upon examination.
- **Changed Bit**: Indicates whether the page in main storage differs from the copy on auxiliary storage. This bit is set by RISC instructions and I/O operations that alter the frame.
- **Pin Indicator**: Indicates that the page is “pinned” in main storage and not eligible to be paged out. This bit is set by SM on request by SLIC routines.

If a frame is unreferenced, unchanged, and unpinned, it is “stolen” (i.e., used for the new request). If it is referred to, it is moved to the back of the available list. Finally, if it is referred to, changed, and unpinned, it is moved to the change list.

3.2.3.3.9.3. Page Out Task

The page-out task writes changed pages back to auxiliary storage. It wakes up when either frames are moved to the change list or at least one task is waiting for allocation of a frame (i.e., a task is in page frame wait state). When not writing pages, it waits on a single task blocker object.

A task enters page frame wait state as follows. If, when attempting to allocate a frame, a task reaches the end of the search queue, it awakens the page-out task, and then waits for available pages.

The page-out task attempts to write pages out from each storage pool’s change list and transfer them back to the available list. After each write attempt, the page-out task wakes up a single user task in page-frame wait for that pool (if one exists). It continues until there are no more waiters and no more pages on the change list for all pools, then it waits on the single task blocker again.

3.2.3.3.9.4. Zero on First Reference

With one exception (described below), MSM guarantees that disk sectors allocated to a segment are always overwritten from main storage prior to their being read (to prevent reading of old data). After copying data from a disk sector into main storage, MSM compares the information in the sector header with the corresponding information in the permanent or temporary directory. If there is a mismatch, MSM clears the main storage frame. This clearing by MSM does not cause the change flag for the frame to be turned on. If the frame should never be written to the disk sector and the frame is removed from mainstore via an explicit remove request or an IPL, a zero on first reference still occurs because the mismatch is still present.

When SLIC components reallocate pages, however, they cannot assume that the pages are cleared by SM when first referred to. A reallocated page is a page that was allocated, truncated, and then re-allocated. In this case, the SLIC routine that requested the re-allocated page is responsible for ensuring that the page is cleared before it is used.

3.2.3.3.9.5. MSM Interfaces

MSM provides interfaces to perform the following:

- Move specified pages between main storage and DASD
- Move the entire contents of an access group between main storage and DASD
- Insert frames into and remove frames from a specified storage pool
- Allocate and deallocate E=R frames of main storage

3.2.3.3.9.6. Tag Management

At the MI, the 8 byte address discussed above is externalized as the address part of a pointer. See section 3.2.1.2.4 “MI Pointers” on page 3-21 for details about pointers. Pointers are protected from misuse or accidental modification by the use of the tag bits previously described in section 3.1.1.1.2 “Tag Bits” on page 3-8. The two tag bits are set whenever the data bits of a quadword (16-bytes) contain a pointer. The tag bits for all quadwords in memory that do not represent a pointer are reset. When an MI object is created, the SLIC creates a system pointer to it and uses RISC instructions to set the tag bits. The tag bits are tested when a pointer value is loaded from a 16-byte area. Attempting to load a pointer that does not have its tag bits set creates an exception that interrupts execution of the program.

Any attempt by the user to modify the contents of a pointer with other than pointer manipulation instructions causes the pointer tag bits to be reset, making the pointer invalid for addressing purposes.

Since the DASD does not support the tag bits, MSM must have a way to save any tag bits when copying frames to DASD and restoring the tag bits when copying disk pages to main storage. This is handled by ANDing the tag bits from each pair of double words to produce a single tag bit per 16 bytes. This results in 256 tag bits per page (one page = 4096 bytes; 16 bytes [per pointer] x 256 = 4096 bytes). The 256 tag bits are stored in the page header.

3.2.3.3.10. Storage Management I/O

SM works the same as other input-output (I/O) in interfacing with physical DASD (see section 3.2.3.7.6 “I/O for DASD” on page 3-95) and is essentially designed the same as other I/O users. The major difference is that none of the SM I/O is externalized to the SM user.

3.2.3.3.11. Start-up and Shutdown

This section describes the actions performed by SM during start-up and shutdown of AS/400.

3.2.3.3.11.1. Storage Management Start-up

The free space maps and the permanent directory may become unusable due to system crashes. At IPL, ASM checks to see if these directories are “usable” (see below). If they are not, then ASM scans the contents of all auxiliary storage and, using information stored in the sector headers and segment headers, rebuilds the free space maps and the permanent directory.

3.2.3.3.11.2. Storage Management Shutdown

Storage management shutdown is a procedure that is executed (if possible) before machine processing is terminated. It might not be possible to perform storage management shutdown when system power is lost (and there is no battery back-up or uninterruptible power supply), the processor fails, the main system bus fails, or other critical hardware fails. The purpose of storage management shutdown is as follows:

- To ensure that all changed frames (pages) currently residing in main storage are written to disk
- To ensure the storage management directories are written to auxiliary storage and marked as “usable” so that the directories do not have to be recovered during the next IPL
- To return all temporary segments to free space

If it is not possible to run the shutdown procedure, either the Main Storage Dump manager or the service processor attempts to dump the contents of main storage to reserved segments on auxiliary storage. This “main storage dump” can then be used during the next IPL to simulate the shutdown procedure.

3.2.3.4. Process Management

Processes are MI constructs that execute on behalf of a user with an associated user profile. They are uniquely identified by a system pointer to a process data structure known as a Process Control Space (PCS). They are associated with a user by the system pointer to the user profile contained in the PCS. Every MI process has an associated SLIC task. The task-dispatching element structure (TDE) is maintained as part of the process data structure. More information about tasks and TDE is in section 3.2.3.1 “Tasks and Task Dispatching” on page 3-43.

There are a variety of MI instructions for process management. These instructions are blocked, with the exception of the MI Materialize instructions, which are restricted to the user’s process. Processes are manipulated by the operating system code via MI instructions.

This section discusses the MI process construct, its associated data structures, and program execution.

3.2.3.4.1. Process Initiation

An MI process construct provides an environment in which to run programs on behalf of a particular user. Processes are started as a result of users beginning jobs on the system (i.e., starting interactive or batch jobs, section 4.1.3.3 “Job Management” on page 4-23). Programs run within a process. A user profile is associated with each process when the process is initiated. If the user profile is a member of a group (see section 5.2.1 “Subject Attributes” on page 5-2), then both the user profile and the group profile are associated with the process.

The authority associated with a process can vary during the lifetime of the process via program adoption (see section 3.2.3.5.2 “Program Adoption” on page 3-70) or as a result of modifying the user profile process attributes. When an authority check is performed, the current associated user profile and/or group profile is used as a source of authority to determine access to objects. The authority information is retrieved from the appropriate profile that is associated with the process. Pointers to the user profiles associated with a process are maintained in the PCS.

Processes are established as a result of the MI instruction Initiate Process. The operands of this instruction are the PCS and the Process Definition Template (PDT). The data structure for a process is the PCS, which is created as a result of the MI instruction Create Process Control Space (a blocked MI instruction). A pool of PCSs are created at IPL time, that can be used later as operands of the MI instruction Initiate Process. The PCS data structure is discussed in more detail in section 3.2.3.4.2 “Process Control Space” on page 3-63. A process is initiated from information contained in the PDT.

The MI instruction Initiate Process is issued as a result of the following activities:

- A user signs on to a workstation and begins an interactive job[17] (Subsystem Monitor Job)
- A user transfers to an associated group job (Subsystem Monitor Job)
- A batch job is initiated from a job queue (Subsystem Job)
- A system job is started (IPL System Arbiter Job)

A process is initiated in two sequential parts and executes under two TDEs, that of the initiator and that of the new process. Each process has an associated SLIC task. The data structure associated with a task includes the TDE. Section 3.2.3.1 “Tasks and Task Dispatching” on page 3-43 provides additional information on the task data structures.

The first part of process initiation executes as part of the initiating process (i.e., the issuer of the MI Instruction Initiate Process). During this part, the new process has the same resource attributes as the initiating process. Problems detected during the first part of process initiation are communicated to the initiating process.

The following steps occur as Part 1 of process initiation:

- Validation of the parameter syntax of the PDT
- Initialization of the PCB and first Invocation Control Block (ICB) in the Invocation Work Area (IWA). The IWA provides spaces, managed by SLIC as a stack, that are used for storage required by SLIC and for the ICB for all current program invocations under a process. Sets the first ICB to a standard SLIC routine.
- Initialization of a TDE (including fixing in main storage the first PCS segment containing the TDE and the resident PCB, removing from the storage pool chain, and chaining the TDE to the system TDE chain.) Sets pointer to first ICB, PCB, and state in TDE.
- Validation and locking of the user profile for the new process.
- Transfer of the user profile lock to the new process.
- Enqueueing of the new TDE to the TDQ, enabling the task to start.

The second part of process initiation executes as part of an independent process and operates under the user profile and other attributes defined for the new process, and new process attributes are verified. Any problem detected during this step are communicated to the initiator process. At the successful completion of both initiation steps, the process is said to exist and can then be the target of other process management instructions and process-directed event-signals.

[17] A job is a unit of work on the AS/400 system. Jobs run in subsystems and use system resources, such as memory, disk, and processor cycles, to perform their tasks. Jobs can be assigned priorities and can run either interactively or in batch.

The steps of the second part of process initiation are detailed below:

- Receive User Profile lock and any other locks.
- Complete validation of the PDT. This involves checking the authority of the new process to objects in the PDT, such as the initiation, problem phase, and termination programs.
- Signal a process initiated event or failure event to the initiator process.
- As part of the call to the initiation (or problem phase program if there is no initiation program) set up default system and user domain activation groups for program stack/heap/static work areas. Also set state to state of initiation or problem phase program. See section 3.2.3.4.6 “Storage for Processes” on page 3-66.
- Invoke the initiation and problem phase programs (process is running).
- At the end of problem phase, initiate process termination after normal completion of the specified programs by running the termination phase program and then transferring control back to part 1 to run a SLIC termination routine set at the top of the ICB stack.

The MI-addressable storage for a process is supplied by SLIC. This consists of static, heap, and stack.

The initial state of a process is based on the current state of the initiating process. Since the MI Instruction Initiate Process is blocked, the initial state of all processes is system state.

3.2.3.4.2. Process Control Space

The PCS is an MI object that is used by OS/400 to control process execution. OS/400 must create a PCS before the initiation of a process. The PCS is then associated with that process for the entire period that the process exists. The PCS is used for work areas and storage related to process execution. The system pointer to a PCS is used as the target operand when referring to a process in process control instructions and also when signaling process-directed events to a process. The PCS can be used only by one existing process at a time and cannot be destroyed while it is associated with a process (due to a lock). A PCS is associated with a process when it has been specified as the PCS operand on an MI instruction Initiate Process. When additional storage is required during execution of a process, the system extends the PCS.

As illustrated in Figure 3-17. “Process Control Space” on page 3-65, the PCS and its supporting structures consist of the following parts:

- Encapsulated Program Architecture (EPA) Header in the PCS base segment

  This is the structure that is at the beginning of every MI object.

- Non-resident Process Control Block (PCB) in the PCS base segment

  This contains some of the data fields that are required to manage the process or data that is tracked on a per process basis. The term *non-resident* refers to the fact that this structure is pageable.

- An E=R Page

  Defines a page that is allocated “effective address=real address”. This means that the page only exists in mainstore and is never pageable. It contains the following:

  - Task Dispatcher Element (TDE)

    This area contains all of the information needed to control the task or information that is tracked on a task basis.

  - Resident PCB and Process Definition Template (PDT)

    This contains some of the data fields that are required to manage the process (task) or data that is tracked on a per process basis. The term *resident* refers to the fact that this storage is guaranteed to be in mainstore and that a page fault will not occur when accessing it.

  - Resident Light Weight Process Control Block (LWPCB)

    This contains some of the data fields that are required to manage the lightweight process (task) or data that is tracked on a per task basis. The term resident refers to the fact that this storage is guaranteed to be in mainstore and that a page fault will not occur when accessing it.

- An Invocation Work Area (IWA) Segment

  This segment contains the following:

  - ICB Stack

    This area contains the stack frames for MI and SLIC invocations.

  - Stack Control Block (SCB)

    This structure contains fields that are used to control the ICB Stack or contain information about the ICB stack.

  - Non-resident LWPCB

    This contains some of the data fields that are required to manage the lightweight process (task) or data that is tracked on a per task basis. The term *non-resident* refers to the fact that this storage is pageable.

- A segment allocated to track the address of temporary space objects on a per-process basis.

- A segment allocated for recycling process segments.

- Tombstone segments

  Segment(s) allocated by Activation/Invocation (AI) for containing tombstones. Tombstones are used to create alias addresses exposed to the MI. They add a level of indirection such that existence of the target can be checked before access.

- Process Activation Work Area (PAWA)

- PCS associated space containing the Process Communications Object (PCO) and the Name Resolution List (NRL) pointers

- Process storage spaces for stack, static, and heap

The non-resident PCB contains a system pointer to the user profile governing the process execution and an indicator of group profile.

The IWA segment is internal storage used as the basic stack for invocation of programs running in the process. Each such allocation is called an invocation control block (ICB). The ICB is used to save registers and to provide a work area. An ICB for an MI program invocation is also used to describe attributes associated with the invocation and hold items such as exception descriptions and space pointer machine objects. Bits in the ICB indicate the following conditions regarding program adoption:

- Whether the user profile is adopted

- Whether adoption is propagated to further invocations

- Whether the user profile is propagated from a higher invocation

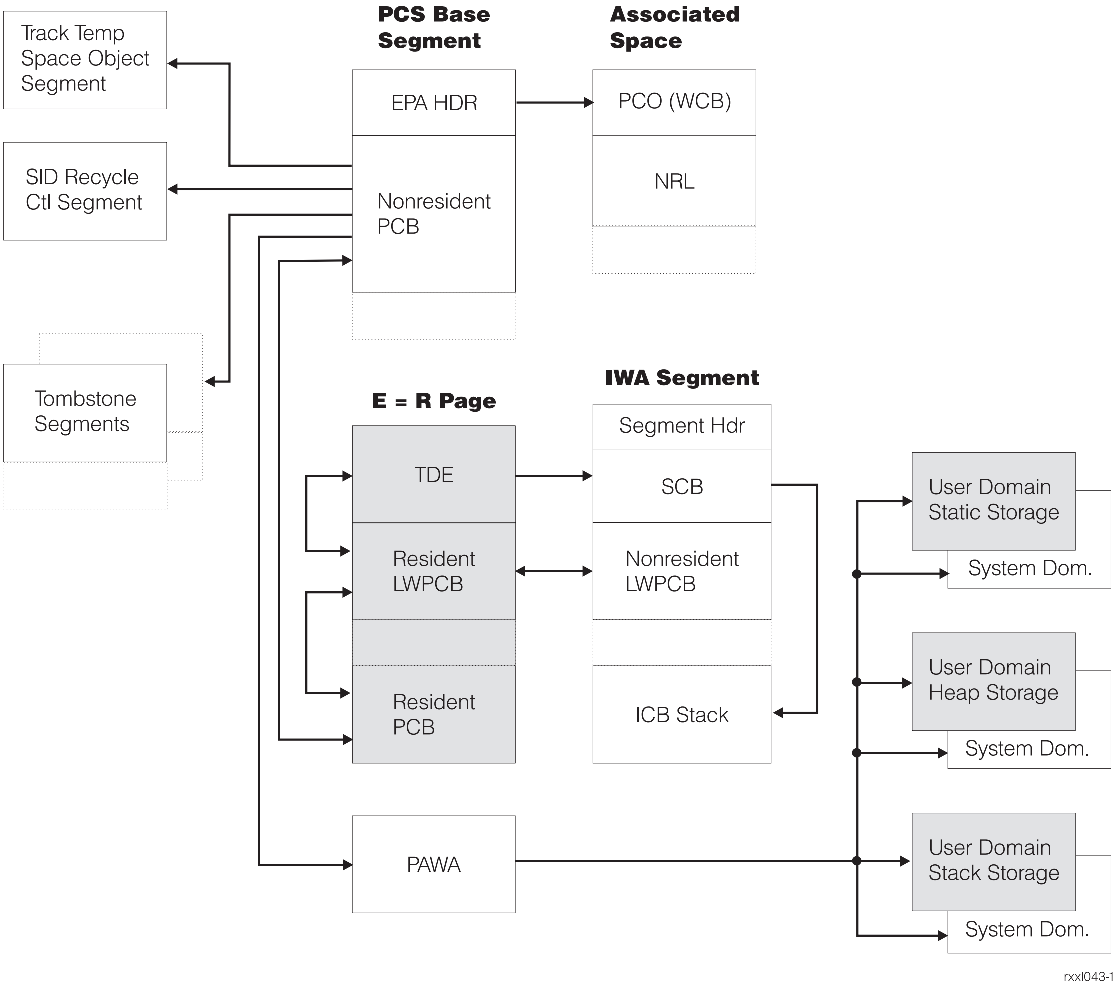


Figure 3-17. Process Control Space

See section 3.2.3.5 “Program Activation and Invocation” on page 3-67 for further discussion of ICB and ICB initialization. See section 3.2.3.5.2 “Program Adoption” on page 3-70 for a discussion of program adoption.

The non-resident PCB holds information regarding process attributes and pointers that link the other portions of the PCS together. It also ties the rest of the structure to the TDE. The PCB contains pointers to the following other areas:

- The Activation and Invocation (AI) anchor
- The Name Resolution List (NRL)
- The first Invocation Control Block (ICB)

The process communications object (PCO) is a space that is used by OS/400 to contain the Work Control Block (WCB). OS/400 programs access the WCB primarily to get pointers to the operating system entry point table and to other work areas in OS/400.

The NRL is used to resolve system pointers to MI objects. It specifies a list of contexts to be searched to locate an MI object. OS/400 stores the NRL in the associated space of the PCS. Whenever the MI instruction Initiate Process initiates a process, a space pointer specifies an NRL. This space pointer points to a region in a space that contains a count and a vector of system pointers to contexts. The count indicates the number of system pointers to contexts contained in the NRL. The contexts are searched in the same order as they are specified in the vector.

The MI instruction Destroy Process Control Space causes the specified PCS to be destroyed and addressability to it to be removed from a context. The PCS to be destroyed must not currently be associated with a process.

3.2.3.4.3. Process Definition Template

The PDT defines the attributes of a process to the MI instruction Initiate Process. These attributes are made up of both pointers and scalars. The pointers address objects such as the user profile, programs to be invoked, and address resolution. The scalar attributes represent resource limits and process control information. The PDT is defined by OS/400.

3.2.3.4.4. Process Termination

Processes are terminated when one of the following occurs:

- Normal return: A process reaches the end of the problem phase.
- Exception termination: An exception occurs that is not handled by the process.
- Terminate instruction: An MI instruction Terminate Process is executed.

3.2.3.4.5. Process State

A process can exist in one of four states:

- Suspended
- Active (running)
- Ready - not active
- Waiting

The state is changed as a result of the operating system issuing blocked MI instructions rather than directly by the user. The user can indirectly affect the state of the process by starting a new job. In this case, the transfer job command would suspend the current process and initiate a new process. The process state is managed by the SLIC as part of task management and is reflected in the position of the task in a queue (i.e., TDQ, wait queue). See section 3.2.3.1 “Tasks and Task Dispatching” on page 3-43 for a discussion of how task state is managed.

A process is suspended as a result of the MI instruction Suspend Process. The execution of a process is resumed as a result of an MI instruction Resume Process.

3.2.3.4.6. Storage for Processes

As described earlier in process management, the PCS is created from above the MI, and the PCS and its associated space are managed by job management (see section 4.1.3.3 “Job Management” on page 4-23 for details). Besides the PCS, many other spaces are needed to invoke and execute a series of programs in a process such as the PAWA, heap, and static storage. Since system state and user state programs each run in separate activation groups, all working storage allocated to programs running in system state is separated from storage allocated to programs running in user state and protected by being in system domain.

This section discusses allocation and reuse of segments within a process. The SLIC allocates pools of storage beginning at IPL that are available to process management and activation invocation to reduce SID consumption. These pools are managed by a process segment SLIC routine that is invoked through a macro named PROCSEG. The PROCSEG routine provides functions to allocate, deallocate, recycle, and destroy process segments.

These segments, though allocated as normal segments by storage management, are, from an MI perspective, neither space objects nor associated spaces. They are simply process local working storage controlled by SLIC. The MI process spaces are addressable via space pointers created by SLIC and restricted to the segment.

These storage segments are divided into three categories: SLIC only addressable, MI-addressable system-domain, and MI addressable user-domain process space. Lists of SIDs associated with destroyed process segments are maintained for each of these categories. These lists are managed as last-in-first-out (LIFO) in which segment addresses are taken from or added to the appropriate queue, with the exception of MI user domain segments, as appropriate. If a list is empty, the PROCSEG routine invokes storage management to create another segment of the required type. The SIDs associated with a destroyed process segment are all recycled within a process by category. The SLIC-addressable and MI-addressable system domain process SIDs are recycled on a system basis when a process terminates. The MI-addressable user-domain process SIDs that are not identified by a user profile are not reused. Those SIDs identified by a user profile are saved on the list and recycled to other processes running under the same user profile.

To prevent processes from being limited to segment-allocated or number-defined size, each component that uses segments from PROCSEG monitors for both exceptions that indicate the allocated storage has been used and the end of segment has been reached. In either case, the component allocates more storage or requests another segment from PROCSEG, as appropriate. This is all transparent at the MI.

3.2.3.5. Program Activation and Invocation

Activation is the process of making a program runable, while invocation is executing the program's encapsulated instructions. Activation and invocation provides three primary functions:

- Activation group management
- Preparing a program for execution
- Program and procedure call support

MI program objects execute within a process and may be associated with an activation group. With strong object typing, only objects created as programs can be executed and only in a process. The activation/invocation code checks the target object's EPA header to ensure it is a program object. The attributes of a program include the following:

- State (user/system/inherit)
- Domain (user/system)
- Pointer to owning user profile
- Program adoption flag (run under owner or user profiles)
- Adoption propagation flag (do not use authorities from stack)

Programs that require static storage or have linkage to other programs must be activated before they can be called. Activating a program allocates static storage and further activates any service programs on which the program depends. This makes it possible to resolve addresses to external procedures and data that are defined in other program objects.

Programs that require an activation are associated with an activation group. A system activation group exists for all system state programs and one (OMI and NMI default) or more (user-defined in NMI) activation groups exist for user state programs. The purpose is to ensure that the system heap, automatic, and static storage is separated from users heap, automatic, and static storage. The system storage is in system domain so it may not be accessed by user state programs. NMI users may define multiple activation groups to provide similar separation in their application programs. Activation groups are created implicitly either during process initiation (the process default AGs) or as part of NMI program activation. Activation groups are destroyed explicitly using the MI instruction Destroy Activation Group (DESAGP) or implicitly at process termination.

There are two categories of program objects: OMI programs and NMI bound programs. Both can be executed within the same activation group and process. Each type of program has an associated set of MI instructions for call and return functions. Programs are invoked through call and transfer control MI instructions.

Program activation is responsible for the allocation and initialization of static storage on behalf of the program. OMI programs that require static storage and all NMI programs must be activated. Static storage initialization for OPM programs occurs implicitly as part of a program call (e.g., CALLX, XCTL.) or explicitly if the MI instruction Activate Program (ACTPG) is used. Activation occurs implicitly for all NMI bound or service programs that are bound together as part of a program call to any of the programs bound together or explicitly as the result of an MI instruction Activate Bound Program (ACTBPGM) on any of these programs.

For OPM, each program call checks authority and domain, verifies the parameters being passed, builds and stacks an ICB, and changes the state of the process if required. Initialization of static storage and resolution of the pointer to the program occur when needed.

The NMI activation of a bound program is similar to an OPM, but may involve more process spaces being created, depending on the complexity of the program being invoked. For example, if the program option specifies a new activation group, when called, the CALLPGM routine creates a new activation group. However, the advantage of the NMI bound programs is the initial binding of service (library routines) programs, including DAC, which is accomplished when the base bound program is activated. The MI instructions Call Bound Procedure or Call Procedure Pointer are used for making the linkages to service programs.

Linkage data is held in encapsulated storage, which is accessed either by SLIC or by translator generated code as part of call sequences. There are two types of linkage information, which are held in different encapsulated structures. The MI program cannot obtain a space address of these structures.

Dynamic linkage data (or control flow) is held in the Invocation Stack Frame (ISF), which is maintained in the encapsulated control stack. For example, register state and return address are stored in the register save area of the ISF. The ISF is access by SLIC code or translator generated code.

Static linkage data is held in encapsulated data structures known as Module Binding Vectors (MBV) associated with a program activation. Static data includes the addresses of external data and procedures that a program references. Linkage data is stored in 8-byte address format. These structures are referenced by SLIC code and by translator generated code. For example, to perform a procedure call, the call sequence obtains the address of the target procedure from the encapsulated data structure known as the MBV.

The MBV links to external procedures and static data. NMI programs use inline code to effect procedure calls. NMI programs may be bound to other service programs. This binding is symbolic in nature. For example, if program A calls procedure X, which is exported by service program B, then the export ID (the index into the service program’s export list) of X is recorded in program A along with a symbolic reference to service program B. During program activation, all such references are converted to addresses. This results in the activation of not only the primary program, but all service programs referred to transitively by that program.

Automatic storage is supplied to MI programs from a separate stack known as the *automatic storage stack*, not the control stack. Each activation group has it’s own automatic storage stack. The automatic storage stack contains neither control flow information nor 8-byte addresses. The MI program has space addressability to the automatic storage stack.

The control blocks needed to activate a program are managed in a SLIC encapsulated space called the process activation work area (PAWA).

This is a user-domain space containing all the structures needed to maintain the activation group, including the PACBs, MBVs, and lists of static and heap storage. It is user domain so the activation of user state programs can be accomplished without changing state. There are no pointers to this area above MI and no chain of pointers that allow a user to gain access.

The program state of an MI program object determines two things:

1. The protection domain of the MI storage (auto, static, heap) provided by SLIC. Domain protection for MI storage is implemented using hardware storage protection (HSP). It is the use of system domain protection for MI storage that protects OS/400 data structures from corruption by users.
2. The process execution state in which it runs when called. SLIC sets the state in the MSR based on the encapsulated program state and the current MSR state (in the case of inherit state programs). There is nothing an MI program can do itself to alter the process execution state in which it runs. The process execution state is set only by SLIC code. Saved values of the MSR are saved only in encapsulated storage.

3.2.3.5.1. Call and Return

There are two types of call mechanisms provided by the MI. However, both forms operate similarly from a low-level linkage perspective. State information, in the form of return address, caller’s stack frame and non-volatile register contents is stored in a frame known as an ISF on the encapsulated control stack. The prologue code of an MI procedure stacks a frame and saves the necessary state information; the epilog code restores state information from the ISF, unstacks its frame, and returns to its caller.

Program calls. The target program is identified by a system pointer. The usual DAC mechanisms are employed to ensure that the caller is authorized to call the program. SLIC code is always required to effect a program call. A program call transfers control to a pre-determined entry point within a program called the Program Entry Point (PEP).

Procedure calls. Procedure calls are a lightweight call mechanism that can transfer control to procedures within the same program as the caller or to a different program. The target procedure is identified either statically by a symbol name or dynamically by an MI procedure pointer.

A static name is resolved during binding time. The binder determines the symbolic name of the program that exports the procedure. The calling program records the program name and a symbolic identifier for the exported procedure. We say that the calling program is “bound to” the exporting program. When the calling program is activated, all “bound to” programs are also activated. The usual DAC methods are employed to ensure that the process is authorized to the “bound to” programs. The appropriate linkage information is placed in the encapsulated MBV by SLIC activation code. The linkage data is used by the translator generated code for the MI instruction Call Bound Procedure (CALLBP).

A procedure pointer is an MI pointer that contains values of type procedure-address, which can be created only by the SLIC activation code. A procedure-address is created by SLIC in response to an MI program that loads the address of a procedure. For example,

```
extern   void   someFunc();
typedef  void   (*funcP) () fp_t;
                 fp_t fp;
                 fp = &someFunc;
```

In the above example, the program that takes the address of the function “someFunc” is bound to the program that exports “someFunc.” The normal DAC rules apply. Once the procedure-address is stored in the pointer, any program in the process may execute a procedure pointer call using the pointer.

The MI program can assign values of the correct type to a procedure pointer but it cannot manufacture a procedure-address value. For example, it cannot cast a numeric value into a procedure-address. The following does not work on an AS/400:

```
fp = (fp_t) 0x1234567890;
```

A call via a procedure pointer uses the encapsulated information in the pointer. In certain cases, the procedure pointer call must be carried out by SLIC “glue” code. This is required when the process execution state must be changed. In either case, the integrity of the pointer is verified before the call sequence is executed.

Programs that are called terminate by either transferring control and being removed from the stack, incurring an unhandled exception (see section 3.2.3.8 “SLIC Exception Management” on page 3-96), or using the return MI instruction. The return is the normal way to terminate a program.

3.2.3.5.2. Program Adoption

A process can gain additional authority via invocation of a program that adopts the program owner’s authorities. The MI program object has an attribute in its header to indicate that while the program is executing, the process has available the authority of the program owner’s user profile in addition to the other user profiles associated with the process. The user profile that owns the program can be adopted for use by the process as a supplemental source of authority. The adopt-owner-authority bit is set when the program is created and may only be changed using Change Program CL Command.

A program, by default, runs with all authorities of programs on the stack that adopt owners’ authorities.

In addition to the adopt-owner-authority bit in the program header, another bit, known as the propagate bit, is defined. This bit stops or allows the use of any adopted authority from above it on the stack.

A process can gain additional authority through program adoption in the following steps:

- When a program is created, the creator can specify that the program is to have the adopted user profile attribute.
- When the program is invoked, the user profile is adopted by the process, and the authorization rights of the owner of the program are available to the process as long as the program is on the stack.
- When a program is created with the adopted-user-profile attribute, this same program can also have the propagate-adopted-user-profile attribute. When a program with these two attributes calls another program, its adopted-user-profile authorities are also available to the called program as well as any additional called programs. All user programs are created with the propagate-adopted-user-profile attribute. Only IBM internally created programs can be created without the propagate-adopted-user-profile attribute.
- For byte stream file[18] and directory objects, the Integrated File System object manager does not check adopted authority to authorize access to these objects.

[18] This includes external objects which use an MI byte stream file for these implementation, such as, stream files (*STMF) and symbolic links (*SYMLNK)

The program adopted-user-profile's authority is available to succeeding invocations of other programs unless one of the following conditions is true:

- the calling program's propagate-adopted-user-profile attribute indicates no propagation[19]
- the called program's use-adopted-user-profile attribute indicates suppress-previously-adopted-authority[20]
- the calling program uses the MI instruction Call External (CALLX) or Transfer Control (XCTL) and sets the suppress-previously-adopted-authority option
- the calling invocation was changed by the MI instruction Modify Invocation Authority (MODINVAU) to act like the use-adopted-user-profile attribute indicates suppress-previously-adopted-authority

The three bits, adopt-owner, propagate-adopted, and suppress-adopted are stored in the program header by the binder, as determined by the program creator. When a program is activated, the routine that does the activation (the SLIC call code) sets these bits in the ICB. During execution, each time authority is checked, the access control algorithm walks up the invocation stack, checking at each level for adopted authority or a bit indicating no propagation of adopt or a bit indicating that adopted authority should be suppressed. See section 6.2.4 "Access Control Algorithm" on page 6-7 for details about authority checking.

3.2.3.5.3. State Transition[21]

Every program has an associated state (user, system, or inherit). All objects, including programs, have an associated domain (user or system). A state transition can occur when the machine invokes or terminates a program. All MI program objects contain state bits in the program header that determine what state the process will be in when it runs. System and inherit state are set only by using IBM internal tools. They are not a part of user program creation. The binder on an evaluated configuration always sets the state of any program created to user.

The process state transitions are managed by the SLIC call code and SLIC exception management. The state of an executing process can be changed by the following unblocked MI instructions: Call External (CALLX), Transfer Control (XCTL), Return External (RTX), End (END), Call Bound Procedure (CALLBP), Call Procedure Pointer (CALLPP), and Return from Procedure (RET).

The current state of the process is kept in a bit in the Machine State Register (MSR) in the processor. The MSR is a Special Purpose Register, which can be altered only by the privileged RISC instructions mfmsr and mtmsr. The SLIC activation invocation and exception management components use these instructions to manage the state of the process for each MI invocation. The state of the previous invocation is kept in each Invocation Stack Frame (ISF) for correctly restoring states as the stack is collapsed.

When a task is swapped out, the contents of the MSR are saved in the Fixed Register Save Area Object of the TD Task Object. When a task is dispatched, the contents of the MSR that was saved are loaded into the MSR in the processor, thus restoring the state (user or system) to what it was when the task was swapped out.

[19] Only the calling program's adopted user profile authority is not propagated. Any adopted authority propagated by other programs on the invocation stack is still propagated to the called program.

[20] The suppress-previously-adopted-authority function suppresses (prevents the propagation) of all adopted authority from any other program on the invocation stack to the called program.

[21] The state transitions discussed in this section refer to the process execution state (user or system) and not to the overall process state (suspended, active, ready, waiting).

3.2.3.6. SLIC IPC Management

Communication between tasks has been discussed in section 3.2.3.2 “Queueing Objects” on page 3-45. Communication also occurs between MI processes at the SLIC level. This communication can occur at MI process initiation time or during MI process execution.

At process initiation time, the argument parameter list, which is specified as an operand on the MI instruction Initiate Process, can be used to pass information between processes. In addition, the MI instruction Initiate Process can also pass a space pointer to a space MI object called a process communication object (PCO). The MI Instruction Materialize Process Attributes can be used by the initiated process to access the PCO. The PCO’s format must be agreed upon by the communicating processes; like any other space MI object, it is manipulated by the set of MI instructions that use space pointers to operate on space MI objects.

During MI process execution, interprocess communication occurs through queuing and dequeuing of messages and the sending of signals. (Events can also be used to provide synchronization; event management is discussed in section 3.2.3.9 “SLIC Event Management” on page 3-100.) The components that manage IPC are discussed below.

3.2.3.6.1. Queue Management

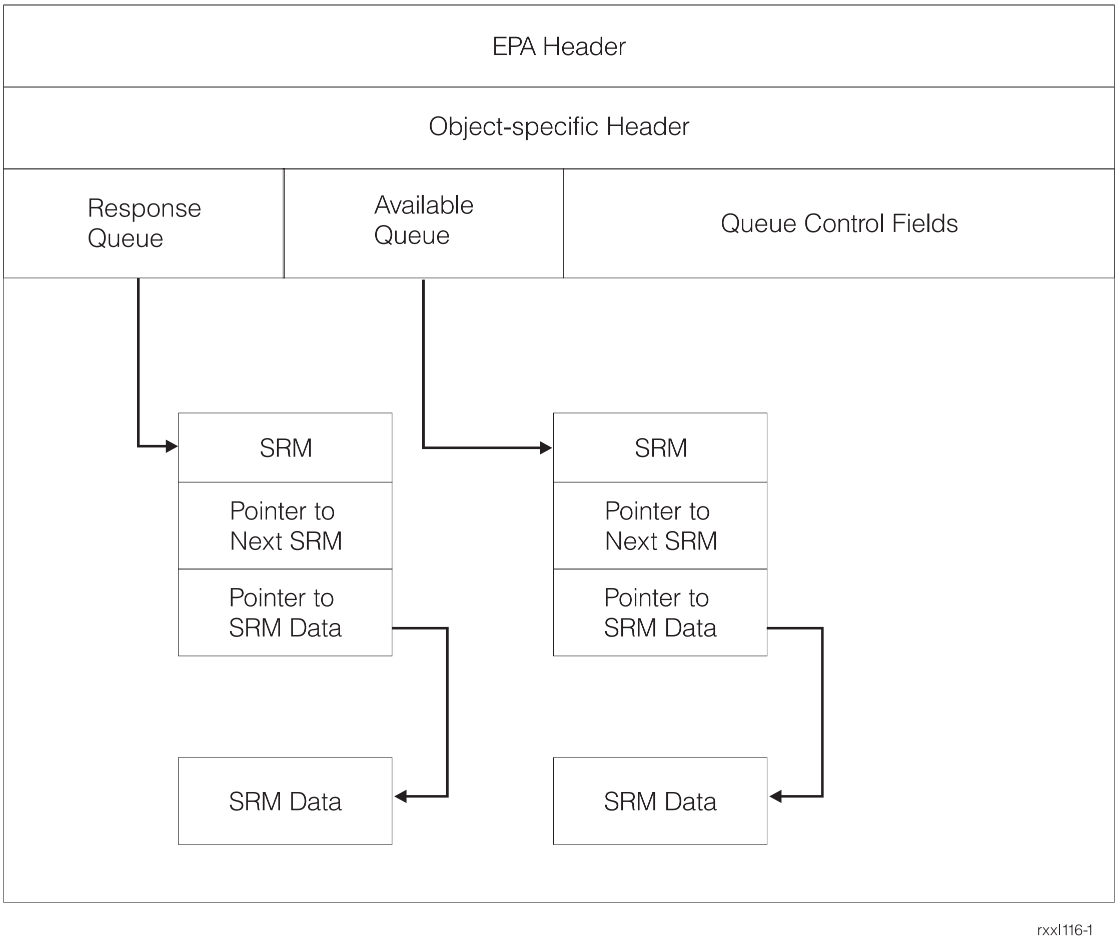


Figure 3-18. Queue MI Object

Queues are MI objects and have been introduced in section 3.2.1.5 “MI Objects” on page 3-26. Figure 3-18. “Queue MI Object” on page 3-73 shows the structure of a queue MI object. The EPA header and object-specific header are followed by two SRQs (SRQs have been introduced in section 3.2.1 “Object Management in

SLIC" on page 3-17). The first SRQ is called the Response Queue, and the second is called the Available Queue. After the two SRQs are the control fields for information such as the following:

- Maximum number of messages
- Message text length (must be less than 64 KB)
- Number of message elements available for enqueuing messages
- Whether the queue is of fixed size or is extendable
- Queue extension length, if extensions are possible
- Storage area allocated in excess of that requested when the queue was created
- Ordering sequence to use for messages: last-in-first-out (LIFO), first-in-first-out (FIFO), or keyed
- Key length, if a key is present
- Current IPL number (used for recovery processing, if needed)
- Whether message data taken off the queue can contain pointers. More about pointers in messages is found in the discussion of dequeuing below.

These control fields are set when the queue MI object is created.

After the control fields are the message elements. Messages consist of two parts, a message prefix and message text. The message prefixes are SRMs; SRMs have also been introduced in section 3.2.1 "Object Management in SLIC" on page 3-17. Each SRM contains fields for such values as the length of the associated message text and the key, if that is the chosen ordering sequence, and also contains a space pointer to the next SRM on the SRQ, and a space pointer to the associated message text area, which is shown in Figure 3-18. "Queue MI Object" on page 3-73 as "SRM data." As the figure shows, the two parts of a message are not contiguous; the message element is made up of a fixed size SRM and a fixed size message text area.

When the queue MI object is created, all message elements are formatted and enqueued to the available queue. This is possible because the maximum message text length is specified on the MI instruction Create Queue. (A given message's data can be shorter than the specified size, but the message elements for a queue MI object are all formatted to the specified, maximum size.) At this point, none of the message elements on the available queue contain message data, and no messages are enqueued to the response queue.

When a message is to be enqueued, the MI process requesting the enqueue must supply a message prefix and message text. The MI instruction Enqueue uses queueing methods to dequeue an SRM from the Available Queue. It then moves the supplied message prefix into the fields of the SRM and moves the supplied message text into the SRM data area. The supplied message text may contain pointers; if it does, they are preserved by queueing. Next, the SRM is enqueued to the Response Queue.

The QM component includes two component-specific exception handlers (CSEHs) for use when no message elements remain on the available queue. If an attempt is made to dequeue from the available queue, but that queue is empty, then one of two actions occurs:

1. If the queue cannot be extended, a Queue Full exception is thrown.
2. If the queue can be extended, it is extended and one of the new SRMs is returned. In addition, a Queue Full exception is also thrown.

The MI instruction Dequeue causes the SRM to be removed from the response queue and replaced on the available queue. The message text is moved from the message text area to the location identified on the MI instruction. What happens to any pointers in the text depends on the queue control field value. If pointers are allowed in message text, instructions to manipulate tagged pointers are used. Otherwise, the tags are not preserved and the pointers can no longer be used. If keys are used, pointers can be used as key values, but again the tags within the key are not preserved when the message is dequeued.

When the response queue is empty, or, if the messages are keyed and the requested key is not found, the MI process issuing the dequeue can either wait or can exit. If the MI process waits, it can wait indefinitely or for a specified time period. When the time period expires, an exception is raised.

3.2.3.6.2. Queue Space Management

The queue space MI object provided by the QS SLIC component is made up of a base segment, an optional associated space, a heap space, and a machine index. The base segment contains the address of the heap space and the address of the machine index. The heap manager maintains a table that lists the segments comprising the heap space. The machine index maps message indexes to addresses.

The QS SLIC component has three main functions, as follows:

- QS provides a set of macros for use by the SLIC exception handler (EX) component. These macros allow the creation and use of temporary queue space MI objects for use by EX. EX is described in section 3.2.3.8 “SLIC Exception Management” on page 3-96.
- QS manages the process queue space MI object associated with every MI process when the MI process is invoked. A process queue space MI object can contain multiple queues. In addition, QS manages message queues that can be associated with every invocation of a program or procedure within an MI process. One use of these invocation message queues is to queue exceptions presented to a program. The process queue space has been introduced in section 3.2.1.5 “MI Objects” on page 3-26.
- QS provides MI instructions for use by OS/400 components to manipulate queue space MI objects. For example, the message handling OS/400 component uses these MI instructions to build the MSGQ external object from a queue space MI object. See section 4.1.3.7 “Message Handling” on page 4-55.

Within the queue space MI object, each message queue is a double-linked list of messages. The linking and unlinking of messages is done by QS. Messages can be variable length. No SRQs (or associated SRMs) are used.[22] The messages are stored in heap storage; QS allocates storage for the message, or, when the message is destroyed, returns storage to the heap.

3.2.3.6.3. Signal Management

Signals are a POSIX defined mechanism for interprocess communications. The POSIX.1 standard defines a signal[23] as “a mechanism by which a process may be notified of, or affected by, an event occurring in the system.”

A signal is said to be *generated* when the event (phenomenon) that causes the signal first occurs. Examples of such events include machine detected errors, timer expiration, and terminal activity. A *synchronous* signal is a signal that is generated by some action attributable to program execution within the process, such as a machine detected error. An *asynchronous* signal is a signal that is generated for the process by an asynchronous event, such as terminal activity or an expired timer.

Each process has an action to be taken in response to each signal defined by the system. A signal is said to be *delivered* to a process when the specified signal handling action for the signal is taken.

A signal is said to be *pending* during the time between the generation of the signal and its delivery. Ordinarily, this interval cannot be detected by an application. However, a signal can be blocked. When a signal is blocked, the signal handling action associated with the signal is not taken and the signal remains pending

[22] The message structure provided by the QS component differs from the message structure provided by the QM component. Therefore, the term “message” is ambiguous unless the component handling the message is known.

[23] ISO/IEC 9945-1:1990, clause 2.2.2.81, page 19. Institute of Electrical and Electronics Engineers, INC., New York, 1990.

until it is either unblocked or the signal action associated with the signal is set to ignore the signal. The signal blocking mask defines the set of signals that are blocked from delivery to the process. The signal blocking mask is maintained on a per process basis and is inherited from the parent process.

The *signal action vector* is a list of signal handling actions for each defined signal. The signal action vector is maintained on a per process basis and is inherited from the parent process. The signal action vector specifies the signal handling actions for both synchronously and asynchronously generated signals.

The machine supports a maximum of 63 signal monitors per process. The number of in-use signal monitors is set by OS/400. OS/400 also specifies the attributes for each in-use signal monitor, which consist of a signal number, signal action, signal default action and maximum number of signals to be retained. In addition to the signal monitor attributes, OS/400 also specifies an initial signal blocking mask. The number of in-use signal monitors, signal monitor attributes and signal blocking mask is referred to as *signal controls*. The system signal controls are set through the MI instruction Modify Machine Attributes (MODMATR).

A signal can be generated or delivered only to a process that has expressed an interest in signals. An error condition results if an attempt is made to generate a signal for a process that has not been enabled for signals. A process can express an interest in signals through the MI instruction Initiate Process (INITPR) or the MI instruction Modify Process Attributes (MODPRATR).[24] In addition, the MI instruction Modify Signal Blocking Mask (MODSIGBM) or Modify Signal Monitor (MODSIGMN) implicitly enables the process for signals.[25] If the process has not been enabled for signals, the process signal controls are initialized from the system signal controls stored in the machine attributes (see above). An error condition results if an attempt is made to enable signals for the process when the system signal controls have not been initialized (set) in the machine attributes.

Once the process signal controls have been initialized, the user is permitted to modify the signal controls for the process. For example, the signal blocking mask[26] and the signal action[27] for a signal monitor are commonly modified. Some signal controls, such as the number of in-use signal monitors and the signal default action for a signal monitor, cannot be modified at the process level.

The *signal monitor* defines the actions to be taken when a signal is generated for the signal monitor. The signal monitor contains the following information:

- Signal number

  The signal number is for use by OS/400 and has no significance to the machine. The mapping of signal numbers to signal monitors is done by OS/400. Signal numbers are provided for portability since other platforms associate signal actions with signal numbers. The signal number for a signal monitor is set in the system signal controls and cannot be modified for a process signal monitor.

- Signal action

  The signal action defines the action to be taken by the machine upon receipt of an unblocked signal by the process. The user is allowed to modify the signal action for a process signal monitor. The possible signal actions are as follows:

  – Handle using signal default action

    Indicates that the machine is to take the action specified by signal default action value when the signal is eligible to be scheduled.

[24] A user can explicitly express interest in signals by calling the Qp0sEnableSignals() API.

[25] A user can implicitly express interest in signals by calling APIs, such as sigaction(), that issue the MODSIGBM or MODSIGMN instructions.

[26] A user uses the sigprocmask() API.

[27] A user uses the sigaction() or sigsuspend() API.

  - Ignore the signal (discard)

    Indicates that the user is not interested in handling the signal. When an ignored signal is generated for the process, the machine automatically discards the signal, regardless of the blocked/unblocked state of the signal monitor.

  - Handle the signal by executing signal catching function

    Causes the machine to invoke the signal catching function when a signal is received for the signal monitor. The signal catching function is set to point to a procedure within an activated activation group.

- Signal default action

  The signal default action defines the action to be taken by the machine when the signal action is set to “handle using signal default action.” The signal default action for a signal monitor is set in the system signal controls and cannot be modified for a process signal monitor. The possible signal default actions are as follows:

  - Terminate the process

    Places the process in termination phase, allowing invocation exits to be invoked. If the process is already in termination phase, this action is ignored.

  - Terminate the request

    Cancels all invocations up to the nearest invocation that has an invocation status of request processor. If an invocation with an invocation status of request processor is not present, the “terminate the process” action is taken.

  - Ignore the signal (discard)

    Causes the machine to discard the signal. A signal is discarded for a signal monitor in the blocked state when the signal action is handle using signal default action and the default signal action is ignore the signal (discard).

  - Stop the process

    Causes the machine to place the process in the stopped state. When a process is in the stopped state, execution of the process is temporarily suspended until a signal is generated for the process that has “continue the process if stopped” as its signal default action. When a process is in the stopped state, the normal process control functions remain in effect (the process may be suspended, resumed, or terminated). When a signal is generated for a signal monitor that has stop-the-process as its signal default action, the machine removes any pending signals for signal monitors that have continue-the-process-if-stopped as their signal default action.

  - Continue the process if stopped

    Causes the machine to resume execution of the process that is in the stopped state, even if the signal monitor with the signal default action of “continue the process if stopped” is in the blocked state or has a signal action of “ignore the signal (discard).” When a signal is generated for a signal monitor that has “continue the process if stopped” as its signal default action, the machine removes any pending signals for signal monitors that have “stop the process” as their signal default action.

  - Signal exception

    Causes the machine to signal the “asynchronous signal received” (hex 4C03) exception to the process. The signal monitor number, as well as signal-specific data, is included in the exception-related data.

- Maximum number of signals to be retained

  Indicates the number of signals that the machine retains when the process is masked for signals or when the signal is blocked. While this number of signals is pending, signals received are discarded.

- Current number of pending signals

  The number of signals that have been generated for this signal monitor whose signal handling action has not been scheduled.

- Signal options

  An additional set of attributes for the signal monitor. The signal options primary use is to specify an additional set of actions to be taken by the machine when a signal catching function is invoked.

- Signal priority

  The relative importance of this signal compared with other signals being monitored within a process. The signal priority establishes the order in which signal handling actions are scheduled if multiple signal monitors have been signaled. Signal handling actions are taken in the order the signal monitors were signaled when multiple signal monitors have the same priority. The signal priority default value is 128.

A signal is generated by sending a request with the MI instruction Send Signal (SNDSIG)[28] to a signal monitor. Scheduling of the signal handling action is controlled separately for each signal monitor through the signal blocking mask. The signal blocking mask is a bit mask that defines the set of signals to be blocked from delivery to the process. The blocked or unblocked option specified for the *nth* bit position in the signal blocking mask is applied to the *nth* signal monitor defined for the process.

The process to receive the signal is identified by a process ID.[29] The process ID is used to indicate whether the signal should be sent to an individual process or to a group of processes (known as a process group).[30] The process ID is a 4-byte number that is used to locate an entry in the system managed process table. A process table entry contains the following information relating to the process:

- Parent process ID
- Process group ID
- Status information

The parent process is the logical creator[31] of the process. A process group represents a collection of processes that are bound together for some common purpose. An error condition results if the process ID specified when a signal is sent does not represent a valid process or process group.

The process sending a signal must have the appropriate authority to the receiving process. The initiating process always carries implicit authority to send a signal. The parent process also carries implicit authority to send a signal to a child process. (The parent process ID of the receiving process is equal to the process ID of the process sending the signal.) A child process is allowed to send a signal to its parent process. (The process ID of the receiving process is equal to the parent process ID of the process sending the signal.) Any other

[28] The user uses the kill() API to send a signal.

[29] Only processes that have expressed interest in signals or have used certain UNIX-type APIs (such as spawn()) are assigned a process ID and have an entry placed in the process table. The MI instructions Modify Process Identifiers (MODPID) and Materialize Process Identifiers (MATPID) are used by OS/400 to assign process identifiers and maintain the parent, child, and group relationships.

[30] A process is assigned to a process group by the spawn() and setpgid() APIs.

[31] The logical creator of a process may not be the process that actually initiated the process. For example, if process A uses the spawn() API to request the creation of a new process B, the process B is actually initiated by an OS/400 subsystem process. Process A is the logical creator of Process B but the OS/400 subsystem process is the actual creator of Process B.

process can send a signal to another process if the process control special authorization[32] is present in the process user profile or in a currently adopted user profile. An error condition results if the process does not have authority to send the signal to any receiving process. [33]

3.2.3.7. SLIC I/O Management

In SLIC, I/O is layered and involves an I/O Hardware Driver, an Inter-Process Communications Facility component (IPCF), the Bus Transport Mechanism (BTM), and the Physical I/O layer (PIO). I/O processing can be either direct I/O or indirect I/O.

At the OS/400 level, an application doing direct I/O issues gets and puts to a device, which receives or supplies the data. Direct I/O can be defined as I/O that occurs because an MI instruction is sent across the MI boundary by an OS/400 application that explicitly requests I/O. Three examples of when direct I/O occurs are as follows:

- An application writing to or reading a tape file
- An application writing to or reading a communications file
- An application writing to or reading a display file

Indirect I/O can be defined as I/O that is requested by the SLIC Storage Management layer. From the OS/400 layer, an external object is being manipulated, and the I/O to DASD is not evident. A page fault occurs when an OS/400 application tries to access a page that is not in main storage. A page out occurs when Main Storage Management writes a changed page back to DASD. Main Storage Management handling of page faults is described in section 3.2.3.3.9.1 “Page Faults and Page Fault Processing” on page 3-58. Main Storage Management handling of page outs is described in section 3.2.3.3.9.3 “Page Out Task” on page 3-59. Three examples of when indirect I/O occurs are as follows:

- An application writing to a space object causes a page fault.
- An application requesting a data-base function can cause a page fault or a page out depending on the function.
- An application’s instructions are executing and a page fault occurs to bring in the next page of instructions.

The following diagram in Figure 3-19. “High-level View of I/O” on page 3-80 shows a high-level, conceptual view of I/O processing in AS/400.

Note: The IOP portion of the diagram is a conceptual diagram showing the functions provided by the IOP. The exact implementation may vary for different IOPs.

The right side of the figure has already been described in section 3.1.1.2.2 “Input/Output Processors” on page 3-12. This section describes the left-hand portion of Figure 3-19. “High-level View of I/O” on page 3-80 below the MI interface, that is, the SLIC I/O layer.[34] The OS/400 layer above the MI is described in section 4.1.3.9 “OS/400 I/O Management” on page 4-59. This section describes the following:

- Inter-Process Communications Facility (IPCF)

[32] This process control special authorization is called “Job Control” (*JOBCTL) special authorization by OS/400.

[33] Child processes are created such that they use the same user profile as the parent process. All of the processes in a process group use the same user profile. Therefore, signals can not be used to communicate between processes with different user profiles unless special authorities are used.

[34] In the following sections, the phrases “in the AS/400” or “on the AS/400” refer to the part of the AS/400 that resides on the processor side of the SPD bus and contains the SLIC I/O Layer. This is to differentiate the functions performed by the SLIC I/O layer from those performed by the Licensed Internal Code (LIC) that executes in a Device IOP.

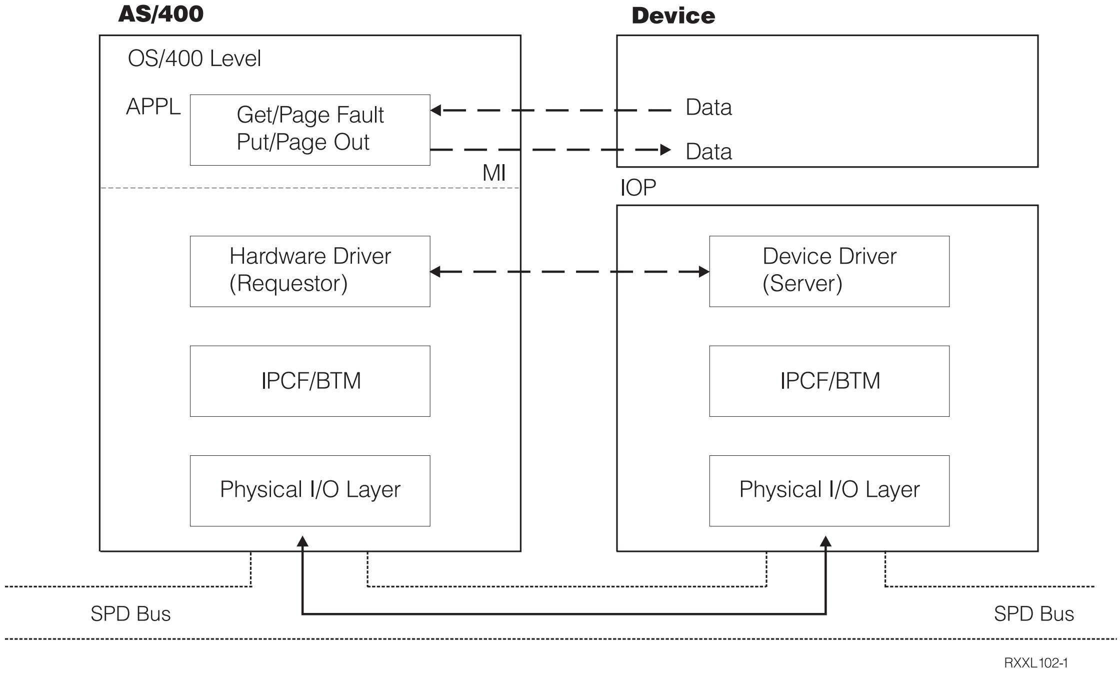


Figure 3-19. High-level View of I/O

- Bus Transport Mechanism (BTM)
- The Physical I/O Layer (PIO)
- The hardware drivers for workstations; printers; tape, optical and diskette devices; and DASD.

I/O for the AS/400 has been divided into direct and indirect I/O above. In addition, direct I/O can be divided into the following:

- Communications - workstations, workstation printers, token ring, ethernet, x25, etc.
- Removable magnetic media - tape, optical and diskette devices
- Other specialized media - encryption IOP.

3.2.3.7.1. Inter-Process Communication Facility (IPCF)

The Inter-Process Communication Facility (IPCF) provides a standard programming interface that allows an I/O Hardware Driver to communicate with a Device Driver. For SLIC I/O, the I/O Hardware Driver is in the AS/400 and the Device Driver task is in the IOP. On the AS/400, the IPCF makes up the SLIC IPCF component. The drivers communicate using requests and responses. One driver makes a request of the other, which carries it out and returns a response. In addition to the request and response, data may travel in either direction between the two drivers.

The I/O Hardware Driver in the AS/400 and the Device Driver in the IOP communicate through a connection. IPCF is responsible, along with the Bus Transport Mechanism, for establishing the connection between the two drivers and for routing the requests and responses over the connection between them. The connection between the two tasks must be established before the sending of any request or response between them. On the AS/400, a Hardware Driver requests the IPCF to open a connection to a specific IOP Device Driver. The IPCF then establishes the connection to the IOP. When the Hardware Driver is finished using the connection, it requests to close the connection. IPCF then breaks the connection between the Driver task in the AS/400 and the Driver task in the IOP.

The I/O Hardware Driver in the AS/400 and the Device Driver in the IOP communicate using exchanges. An exchange is a SLIC encapsulation of a request, response, and data. Outbound exchanges are used for requests sent from the AS/400 to the IOP. Inbound exchanges are used for requests sent from the IOP to the AS/400. The connection keeps track of the active exchanges (request has been sent (or received) but response has not been received (or sent)) and available exchanges (of the maximum number of exchanges supported by the IOP (e.g., 10), those exchanges that are not currently active and are available for use).

There is also an IPCF services connection for each IOP attached to the AS/400. The IPCF services connection has special characteristics that differ from other IPCF connections. It is used by IPCF to establish and service the other IPCF connections between the AS/400 and the IOP. The services connection is an “implicit” connection and is available once the IOP is loaded and operational. SLIC IPCF uses the services connection to do the following:

- Send a request to the IOP to open a new IPCF connection
- Send a request to the IOP to close an existing IPCF connection
- Send a request to the IOP to terminate an exchange on an existing IPCF connection

There is one IPCF service controller for each IOP attached to the AS/400. The service controller keeps track of the connections to the IOP and provides interfaces to reset, dump, and IPL the IOP. There is one service controller task on the system in which all the service controllers run.

The diagram in Figure 3-20. “high-level view of an IPCF connection and exchanges” on page 3-82 shows a high-level view of an IPCF connection and exchanges.

Note: The IOP portion of the diagram is a conceptual diagram showing the functions provided by the IOP. The exact implementation may vary for different IOPs.

3.2.3.7.1.1. Outbound Exchanges

An outbound exchange is a SLIC encapsulation of a request, response, and data for a request sent from the AS/400 to the IOP. The IPCF outbound exchange handles the IPCF part of sending a request from the AS/400 to the IOP and receiving a response from the IOP. IPCF provides the client interfaces to work with the outbound exchange. IPCF works with the Bus Transport Mechanism to build the request, send the request to the IOP, and receive the response from the IOP.

An outbound exchange can have three types of descriptors:

- Request descriptor(s) (required) - A request descriptor describes the request to the server (IOP).
- Data-out descriptor(s) (optional) - A data-out descriptor describes information for the server (IOP) from the requestor (AS/400).
- Data-in descriptors(s) (optional) - A data-in descriptor describes an area in which the server (IOP) may write a response to the requestor (AS/400).

IPCF provides client interfaces to add descriptors to the exchange. IPCF calls Bus Transport Mechanism methods to do the actual adding of descriptors to the exchange. The Bus Transport Mechanism puts the descriptors in a Request Response Control Block (RRCB). An RRCB is used by the Bus Transport Mechanism to control the movement of data between the requestor (AS/400 for outbound exchanges) and server (IOP for outbound exchanges). IPCF also provides client interfaces to send the exchange. IPCF calls a Bus Transport method to actually send the exchange to the IOP.

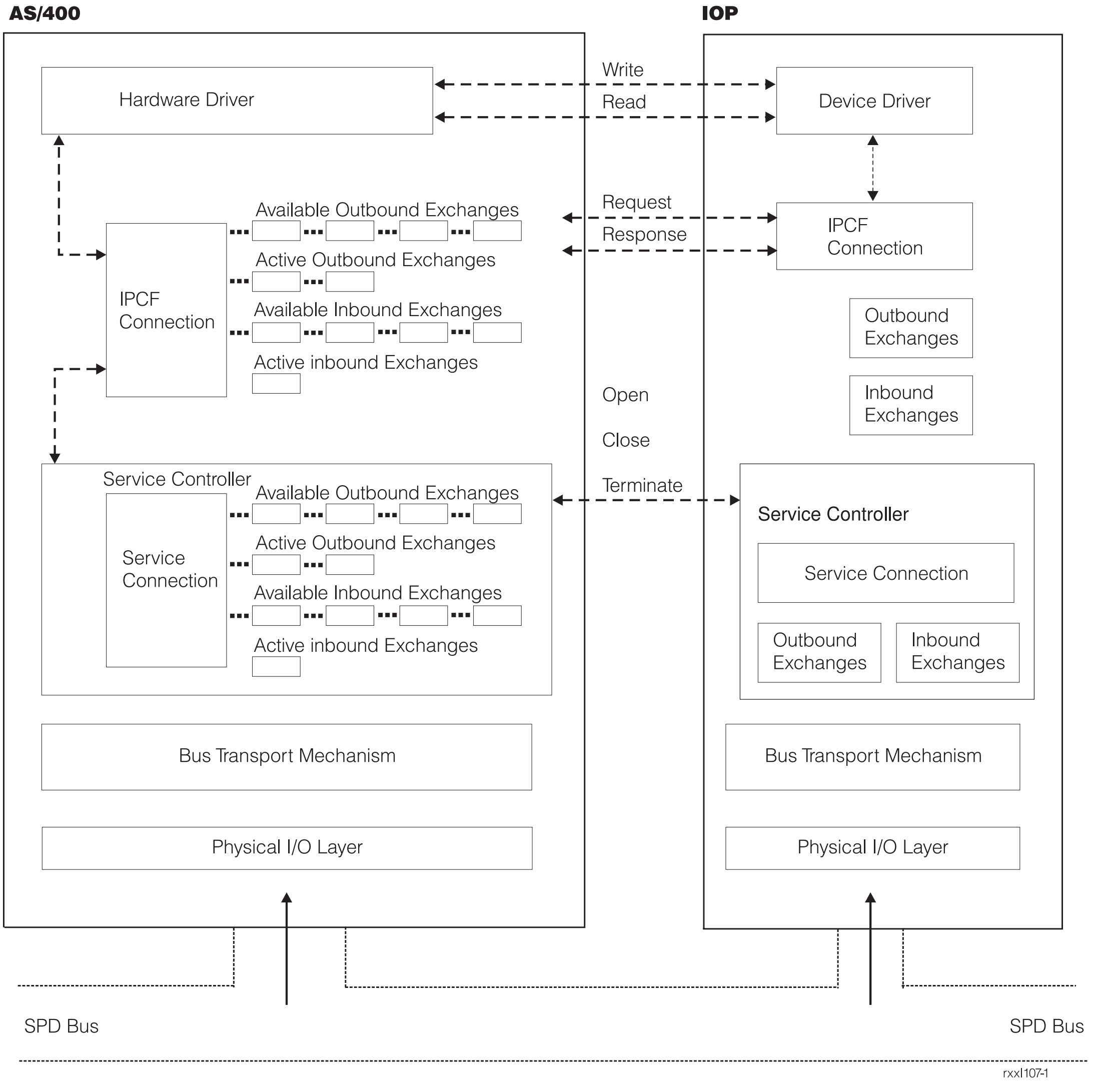


Figure 3-20. high-level view of an IPCF connection and exchanges

The IPCF client provides response entry point code to handle the response sent by the IOP. Some common actions for the response entry point are to send a response note to a queue or call another response entry point to handle the response or issue a return code indicating that a response was received. IPCF is responsible for calling the client response entry point code in the execution context specified by the client when constructing the exchange. The execution context may be one of the following:

- The client's task (or task specified by the client)
- An IPCF router task. Router tasks are used primarily for communications.
- The interrupt handler. Response notifications from the interrupt handler are mainly used by DASD.

3.2.3.7.1.2. Inbound Exchanges

An inbound exchange is a SLIC encapsulation of a request, response, and data for a request sent from the IOP to AS/400. The IPCF inbound exchange handles the IPCF part of receiving the request data from the IOP and sending the response to the IOP. IPCF provides the client interfaces to work with the inbound exchange. IPCF works with the Bus Transport Mechanism to receive the request data and send the response to the IOP.

Like an outbound exchange, an inbound exchange can also have three types of descriptors:

- Request descriptor(s) (required) - A request descriptor describes the request to the server (AS/400).
- Data-out descriptor(s) (optional) - A data-out descriptor describes information for the server (AS/400) from the requestor (IOP).
- Data-in descriptor(s) (optional) - A data-in descriptor describes an area in which the server (AS/400) may write a response to the requestor (IOP).

IPCF provides client interfaces to add response data to an inbound exchange. IPCF calls Bus Transport Mechanism methods to actually add the response data to the exchange. Bus Transport puts the response data in a Request Response Control Block (RRCB). An RRCB is used by the Bus Transport Mechanism to control the movement of data between the requestor (IOP for inbound exchanges) and server (AS/400 for inbound exchanges). IPCF also provides client interfaces to send the response to the exchange. IPCF calls a Bus Transport Mechanism method to actually send the response to the IOP.

The IPCF client provides request entry point code to handle a request sent by the IOP. Some common actions for the request entry point are to send a request note to a queue or call another request entry point to handle the request or issue a return code indicating that a request was received. IPCF is responsible for calling the client request entry point in the execution context specified by the client when constructing the exchange. The execution context may be one of the following:

- The client's task (or task specified by the client)
- An IPCF router task. Router tasks are used primarily for communications.

3.2.3.7.1.3. IPCF Client Interfaces

SLIC IPCF provides interfaces that allow the IPCF client (I/O Hardware Driver) to perform operations on connections and exchanges. For the PLMP clients, macros are provided. For the C++ clients, C++ methods are provided. The interfaces provided include the following functions:

- Open a connection
- Close a connection
- Build an outbound exchange
- Add descriptors to an outbound exchange
- Send an outbound exchange
- Reset an outbound exchange
- Terminate an outbound exchange
- Set response data for an inbound exchange
- Send a response to an inbound exchange
- Receive the request from an inbound exchange
- Receive data from an inbound exchange
- Reset an inbound exchange

3.2.3.7.1.4. IPCF Router Tasks

IPCF router tasks are used to support the PLMP connections that use response entry point and request entry point code provided by the I/O hardware driver. IPCF router tasks are used primarily for communications. If a response is received from the IOP, the router task calls the response entry point containing code written by the client (I/O Hardware Driver) to process the response. If a request is received from the IOP, the router task calls the request entry point containing code written by the client to process the request. The entry point can either handle the response or request or issue a return code telling the IPCF router task to send a response or request note to the client.

The client provides the address of the response entry point code and request entry point code when the connection is opened. The client also provides a client object lock, which the IPCF router task locks before calling the client's entry point to ensure that the client receives the responses or requests in the same order IPCF/BusTransport received them from the IOP. If the client entry point handles the response or request, the client entry point is responsible for unlocking the client object lock after processing the request or response. If the client entry point tells IPCF to send a response or request note, the IPCF router task is responsible for unlocking the client object lock after sending the note to the client.

AS/400 has one system router and a pool of IPCF router tasks. The system router has a queue of messages that need to be handled by IPCF router tasks. When opening the connection, the client provides a number indicating the maximum number of router tasks needed for the connection. If there are not enough unused router tasks to handle the client's open request, IPCF creates additional router tasks and adds them to the pool of router tasks. A count is kept of the number of router tasks processing messages for a particular connection. This count is not allowed to exceed the number of router tasks the client specified on the open. If the connection has reached its maximum allowed router tasks, the message is put on the connection's deferred queue and processed later when a router task is available.

There should be at least one router task running on the system but not more than the number of processors. An additional router task is dispatched if there are more than four messages per router on the system router queue.

3.2.3.7.1.5. Service Controller and Service Controller Task

There is one IPCF service controller control block for each IOP attached to AS/400. The service controller keeps track of the open connections to the IOP and provides interfaces to reset, dump, and IPL the IOP. A service controller for an IOP is created at the end of the IOP IPL processing. The services connection for the IOP is also created when the service controller is created and is contained in the service controller control block.

There is also one service controller task for the system. The service controller task is created early in the IPL of the system when BusTransport and IPCF are initialized. The IPCF service controller task is responsible for the following:

- Opening connections to the IOPs attached to the system
- Closing connections to the IOPs attached to the system
- Terminating exchanges for the IOPs attached to the system
- Handling errors for the IOPs attached to the system

3.2.3.7.1.6. IPCF Control Blocks

IPCF creates several types of control blocks on the system. Some of these control blocks include the following:

- Connection control blocks
- Outbound Exchange control blocks
- Inbound Exchange control blocks
- One system router control block
- Service controller control blocks. There is one service controller control block for each IOP attached to the system.
- Flight recorder control blocks
- IPCF notes. Examples of IPCF notes are request notes, response notes, and close notes.

3.2.3.7.2. Bus Transport Mechanism (BTM)

The SPD Bus Transport Mechanism (BTM) provides a transport mechanism to be used by IPCF to transport data across the SPD bus. The SPD Bus Transport Mechanism defines the set of bus unit messages and data structures for moving data across the SPD bus between the requestor and server. The BTM is unaware of the format or content of any device specific or IOP specific data that is moved between the requestor and server. In the AS/400, the Bus Transport Mechanism is part of the SLIC BusTransport component.

The Bus Transport Mechanism initiates bus traffic as a result of a IPCF client request for work to be done in another processor. The Bus Transport Mechanism in “bus unit A” alerts the Bus Transport Mechanism in “bus unit B” that it has work to do by sending a bus unit message. (If the work is being initiated by the AS/400, bus unit A is the AS/400 and bus unit B is the IOP. If the work is being initiated by the IOP, bus unit A is the IOP and bus unit B is the AS/400.) The bus unit message has enough information for the Bus Transport Mechanism to move the Request Response Control Block (RRCB) into bus Unit B. The Bus Transport Mechanism in Bus unit B can now alert process B that there is work to be done. The process executes the request, which requires data movement between the bus units. The RRCB, which is now in bus unit B’s memory, is used by the Bus Transport Mechanism to control the transfer of data between the bus units.

There are two basic entities used to communicate between the Bus Transport Mechanism in the AS/400 and the Bus Transport Mechanism in the IOP:

- The Bus Unit Message (BUM). A bus unit message is a 12-byte message that initiates all communications between bus units on a SPD I/O bus.
- The Request Response Control Block (RRCB). The RRCB is the control block that is used by the Bus Transport Mechanism in the requestor process and the Bus Transport Mechanism in the server processor to control the movement of data between the processors. An RRCB can have three types of descriptors:
  - Request descriptor(s) (required) - A request descriptor describes the request to the server.
  - Data-out descriptor(s) (optional) - A data-out descriptor describes information for the server from the requestor.
  - Data-in descriptor(s) (optional) - A data-in descriptor describes an area in which the server may write a response to the requestor.

An immediate descriptor contains a copy of the actual request or data. A reference descriptor contains a token representing the address of the request or data. The SLIC BusTransport component provides methods that IPCF can use to add descriptors to the RRCB and retrieve the information from RRCB descriptors.

The diagram in Figure 3-21. “Bus Transport Mechanism from AS/400 to IOP” on page 3-86 shows how the Bus Transport Mechanism is used to send a request from the AS/400 to an IOP.

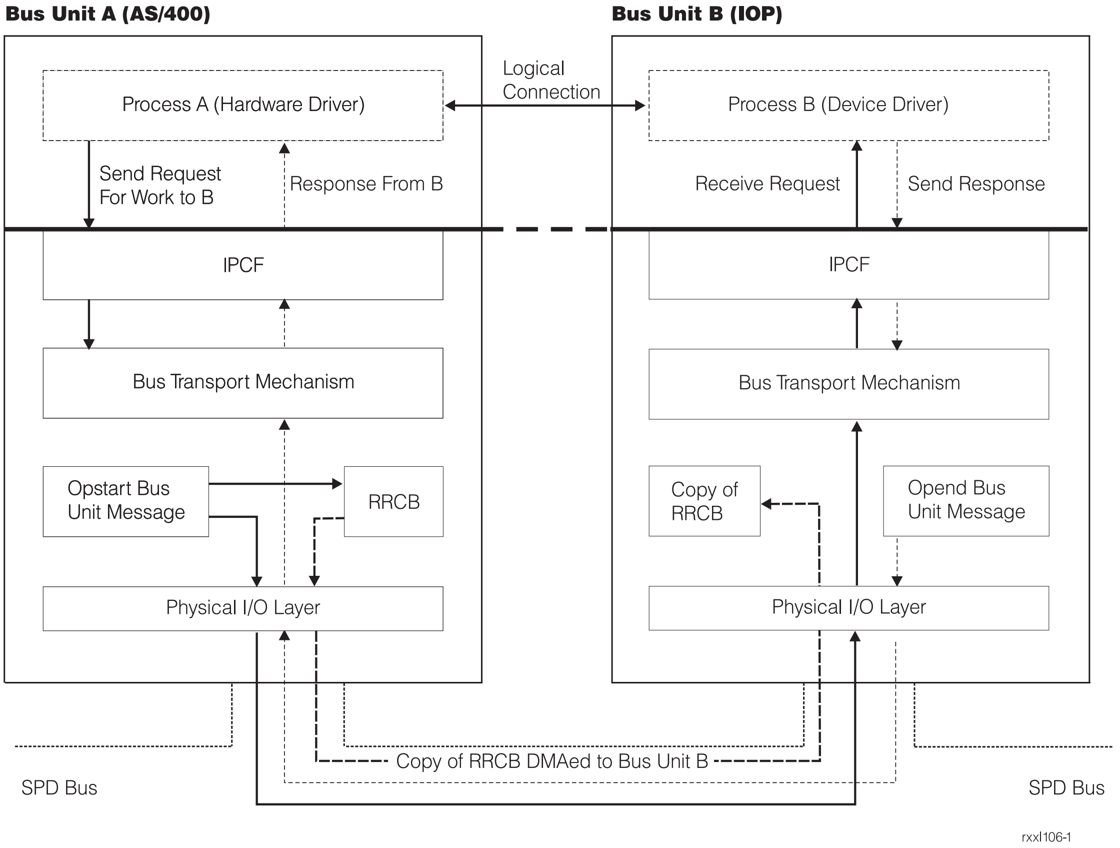


Figure 3-21. Bus Transport Mechanism from AS/400 to IOP

3.2.3.7.3. Physical I/O Layer (PIO)

The Physical I/O Layer (PIO) is the SLIC layer that communicates with the hardware. The PIO is part of the SLIC BusTransport component. See Figure 3-22. “High Level View of Physical I/O Layer” on page 3-87.

On the AS/400, there is a bus object for each bus attached to the AS/400. The bus objects are created when the actual bus is found during the IPL of the system. The bus object contains pointers to the IOP objects for the IOPs attached to the bus.

On the AS/400, there is an IOP object for each IOP attached to the AS/400. The IOP objects are created when the actual IOP is found during the IPL of the system.

On the AS/400, main storage addresses are 64 bits and the SPD Bus can only address 32 bits. Therefore, real addresses cannot be passed across the SPD bus. The SLIC Physical I/O Layer has a Translate Control Entry

(TCE) Table, which is used by AS/400 and the hardware to convert 64-bit addresses to or from 32-bit addresses.

The term SBA refers to the IO hardware used to connect SPD buses to AS/400. The Bus Control Adapter (BCA) is hardware that controls the bus and implements the SBA Architecture. AS/400 software communicates with SBA hardware via Programmed I/O Store and Load commands to certain registers within the SBA and via an I/O Event Stack (IES) and External Interrupt. The AS/400 Physical I/O software initiates I/O bus operations by writing the necessary command information into the appropriate SBA register. The SBA hardware reports status and exceptions by placing information into certain registers within the SBA or by writing an entry into the I/O Event Stack and then sending an External Interrupt to the AS/400 processor. The Bus Control Adapter (BCA) contains information about objects that are shared by the AS/400 and the Bus Control Adapter Hardware, such as the I/O Event Stack and the Translate Control Entry Table.

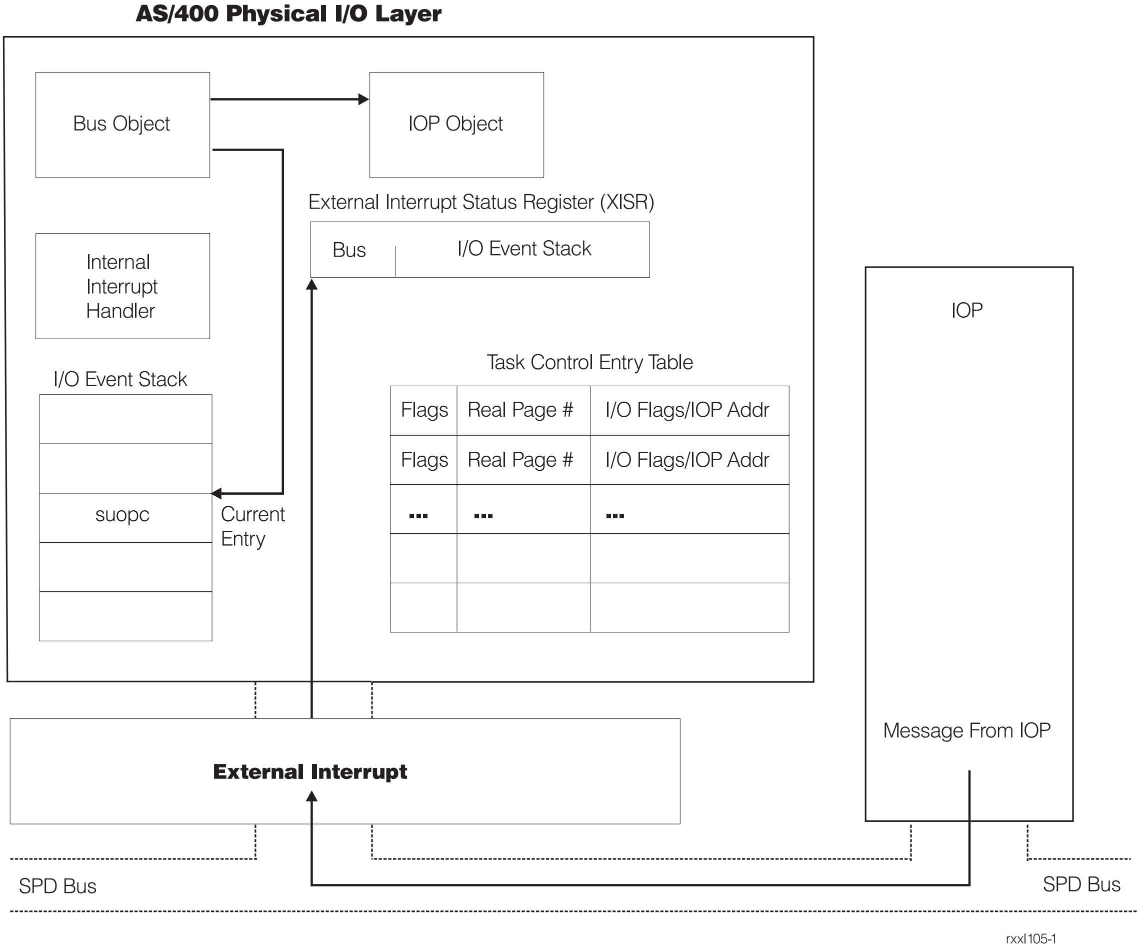


Figure 3-22. High Level View of Physical I/O Layer

See section 3.2.3.7.1.1 “Outbound Exchanges” on page 3-81 for information about how the physical I/O Layer is involved in sending an exchange to the IOP. See section 3.2.3.7.1.2 “Inbound Exchanges” on page 3-83 for information about how the physical I/O Layer is involved in processing an exchange sent from the IOP to AS/400.

3.2.3.7.3.1. External Interrupts, I/O Event Stack, External Interrupt Handler

The SLIC Physical I/O Layer has an I/O Event Stack (IES), which is a circular queue of fixed size entries where the Bus Control Adapter hardware can place completion status or messages. The I/O Event Stack is always written to by the Bus Control Adapter hardware and read by AS/400 software. There is one I/O Event stack for each bus attached to the AS/400.

To inform the AS/400 that a message has been sent to or received from the IOP, the Bus Control Adapter hardware interrupts the AS/400 processor with an external interrupt. An external interrupt is used to inform the AS/400 processor of a message or error status. The External Interrupt Status Register (XISR) shows which bus the interrupt is from and that it is an I/O Event Stack (IES) interrupt. There is one External Interrupt Status Register for each processor on the AS/400. The SLIC Physical I/O layer also has an external interrupt handler, which handles the interrupts and notifies Bus Transport that an interrupt has been received.

3.2.3.7.3.2. Translate Control Entries (TCES) and Tokens

On AS/400, main storage addresses are 64 bits and the SPD Bus can address only 32-bits. Therefore, real addresses cannot be passed across the SPD bus. Instead the IOPs are given a 32-bit token address, which is used in exactly the same way as a real address from an SPD bus viewpoint.

A Translate Control Entry (TCE) is used by AS/400 and I/O to communicate 64-bit main store addresses on a 32-bit bus. The Translate Control Entry contains the following information:

- The real page number
- A flag indicating whether or not I/O is allowed to access the page
- The bus and IOP number of the IOP that is allowed to access the page referenced by this TCE
- Whether or not the TCE is an SPD Bus TCE
- Whether or not write access is allowed to the page referenced by this TCE

There is one Translate Control Entry (TCE) Table containing TCE entries on AS/400. The Translate Control Entry table is stored on the AS/400 but can be accessed by both AS/400 and the Bus Control Adapter Hardware.

A 32-bit token is used to represent a main store address. The token used is 20-bit offset into the TCE table concatenated with the 12-bit page offset for the real address. The Bus Control Adapter (BCA) hardware uses the TCE table offset to get the TCE. From the TCE, the real page number is extracted and concatenated with the 12-bit page offset from the token to get the real address in which to access main store. The Bus Control Adapter Hardware also uses the Translate Control Entry Table to make sure addresses referenced by an IOP are valid.

3.2.3.7.3.3. Main Storage Address Protection

An IOP has the capability to address any part of AS/400 memory via DMA (Direct Memory Access) operations on the SPD I/O bus, but the Bus Control Adapter Hardware (BCA) first verifies whether I/O access is allowed for any given page before going ahead with a DMA operation by doing the following:

- The first check made is an address range check. The Bus Control Adapter Hardware (BCA) checks the token address to determine that it does not point to a Translate Control Entry (TCE) that is beyond the end of the TCE table.
- The second check made is an access authority check. The Bus Control Adapter Hardware (BCA) fetches the TCE entry pointed to by the token address and verifies that I/O access is allowed to the page.

- The Bus Control Adapter Hardware (BCA) also checks that the TCE permits write access if the storage operation is a write to main store and checks the IOBU address of the SPD bus device performing the storage operation against the IOBU address fields in the TCE.
- The third check made is a TCE type check. The Bus Control Adapter (BCA) fetches the TCE entry pointed to by the token address and verifies that it is an SPD Bus TCE by checking the “SPD Bus TCE” flag in the TCE.

If any of the above checks fail, an ISA (Invalid Storage Address) condition is reported.

3.2.3.7.3.4. Sending a Request to an IOP (outbound exchange)

All of the SLIC I/O layers are involved in sending a request to an IOP (outbound exchange). The AS/400 IPCF client (I/O Hardware driver) first builds the exchange and then requests to send the exchange to the IOP using the IPCF client interface. IPCF calls a BusTransport method to send the exchange to the IOP. The Bus Transport Mechanism layer is responsible for building the RRCB and the Opstart Bus Unit Message. The Physical I/O Layer is responsible for sending the Opstart Bus Unit Message on the SPD bus.

The following processing is done to send an exchange to an IOP:

1. The IPCF client (I/O Hardware driver) uses an IPCF client interface to request to send an exchange to the IOP.
2. IPCF calls a BusTransport method to request to send the exchange to the IOP.
3. BusTransport pins the pages of storage referred to by the RRCB reference descriptors and gets the real addresses for the descriptors.
4. BusTransport reserves Translate control entries (TCES) for the RRCB address and the pages of storage referred to by the RRCB reference descriptors. The IOP is given a 4-byte token to represent the address of the RRCB. Each RRCB reference descriptor also contains a token, which represents the main store address of the request or data.
5. BusTransport builds an OpStart Bus Unit Message. An OpStart message is used to alert the server Bus Transport Mechanism that there is an RRCB that requires service. The OpStart message contains a server connection identifier and a token, which represents the RRCB address.

   Note: Some IOPs support Oprequest Bus Unit Messages. An Oprequest message is similar to an Opstart message but the RRCB is contained in the Oprequest message. The Bus Control Adapter (BCA) hardware moves (via Direct Memory Access [DMA]) the Oprequest message into a predefined address in the IOP known as the queue id. The queue id is returned by the IOP when the connection is opened. Opstart and Oprequest messages cannot be used for the same connection.
6. The Physical I/O Layer is responsible for sending messages on the SPD bus. The Physical I/O Layer builds a Start Unit Operation (SUOP) message. The Start Unit Operation message is an SPD Bus Architecture (SBA) instruction used to send a message to an IOP residing on the SPD Bus.

   Note: If an Oprequest message is being sent, the Physical I/O Layer builds a Start Storage Operation (SSOP) message instead of a Start Unit Operation message. The Start Storage Operation message is an SPD Bus Architecture (SBA) instruction used to initiate a DMA operation to an IOP residing on the SPD Bus.
7. The Physical I/O Layer sends the Start Unit Operation (or Start Storage Operation) message. The Bus Control Adapter (BCA) hardware places the message on the bus for the IOP to accept.

   Note: If the software or hardware is busy, the message is placed on a deferred queue to be processed later.
8. The IOP takes the message off the bus and acknowledges back to the Bus Control Adapter (BCA) hardware to indicate that it has received the message.

9. The Bus Control Adapter (BCA) hardware places a Start Unit Operation Complete (SUOPC) on the IO Event Stack (IES) with the bits stating that the SUOP was successful.
10. Note: If an Oprequest message was sent to the IOP, a Start Storage Operation Complete (SSOPC) is placed on the IO Event Stack instead of a SUOPC message.
11. The Bus Control Adapter (BCA) hardware interrupts the AS/400 processor with an external interrupt to notify AS/400 that there is a message to be processed.
12. The SLIC external interrupt handler handles the interrupt and notifies SLIC BusTransport that the interrupt has been received. SLIC BusTransport processes the interrupt and matches up the SUOPC/SSOPC with the outstanding SUOP/SSOP. The Bus and IOP may initiate another operation at this time. Any deferred message(s) on deferred queue are then handled. The IPCF client (I/O Hardware Driver) can perform other work while waiting for the IOP to send the response to the request. SLIC BusTransport notifies the Bus Control Adapter (BCA) hardware that the interrupt has been handled.

See section 3.2.3.7.3.5 “Handling a Response Sent by an IOP” on page 3-90 for information about how the response from the IOP is handled.

3.2.3.7.3.5. Handling a Response Sent by an IOP

The following processing is done to handle a response from an IOP:

1. The IOP uses the Bus Unit Message received (opstart or oprequest) to start performing the I/O operation.

   The IOP is in control of all data sent to or from the AS/400 via Direct Memory Access (DMA) operations on the SPD bus. It sends or retrieves data from the AS/400 system memory to accomplish the I/O operation requested, as necessary. If an opstart message was received, it starts by moving the RRCB into the IOP storage. If an Oprequest message was received, the RRCB has already been moved into the IOP storage. The IOP then uses the tokens provided in the RRCB to send or retrieve the data using DMA operations.
2. After the IOP has finished the OPSTART/OPREQUEST message, it places an OPEND (Operation End) Bus Unit Message on the bus. An OPEND message indicates to the requestor that the server no longer needs access to the data areas specified in the RRCB. The OPEND may also contain completion status.
3. The Bus Control Adapter (BCA) hardware takes the OPEND and writes it to the IO Event Stack on AS/400.
4. The Bus Control Adapter (BCA) hardware interrupts the AS/400 processor with an external interrupt. The External Interrupt Status Register (XISR) shows which bus the interrupt is from and that it is an I/O Event Stack (IES) interrupt.
5. The SLIC external interrupt handler processes the interrupt and notifies BusTransport that an interrupt has been received. SLIC BusTransport matches up the OPEND with the OPSTART/OPREQUEST that was sent.   SLIC Physical I/O notifies the Bus Control Adapter (BCA) hardware that the interrupt has been handled.
6. SLIC BusTransport frees up the resources used to send the exchange to the IOP. This includes releasing the Translate Control Entries used for the exchange and unpinning the pages referenced by the RRCB reference descriptors.
7. SLIC BusTransport notifies IPCF that a response has been received.
8. SLIC IPCF notifies the IPCF client that a response has been received. See section 3.2.3.7.1.1 “Outbound Exchanges” on page 3-81 for information about the various ways that SLIC IPCF can inform the IPCF client that a response has been received.

9. The IPCF client handles the response and resets the exchange so it can be used to send another request to the IOP.

3.2.3.7.3.6. Processing a Request Sent from an IOP (inbound exchange)

All of the SLIC I/O layers are involved in handling a request sent by an IOP to AS/400 (inbound exchange). The Physical I/O Layer is responsible for getting the exchange off the bus and informing the Bus Transport Mechanism layer that there is an inbound request. The Bus Transport Mechanism layer is responsible for getting resources for the incoming request and informing the IPCF layer that there is an inbound request. The IPCF layer is responsible for informing the IPCF client that there is an inbound exchange. The IPCF client is responsible for processing the actual request and sending a response to the IOP.

The following processing is done to send an exchange from an IOP to AS/400:

1. The IOP builds a Request Response Control Block (RRCB) containing the request and any data and moves (via DMA) the RRCB from the IOP to the AS/400 storage so that the AS/400 can access the RRCB. AS/400 cannot directly access anything in IOP storage.
2. The IOP builds an Opstart Bus Unit Message. An OpStart message is used to alert the server Bus Transport Mechanism that there is an RRCB that requires service. The OpStart message contains a server connection identifier and a token, which represents the RRCB address.

   Note: Oprequest messages from IOPs are not supported.
3. The IOP builds a Start Unit Operation (SUOP) message. The Start Unit Operation message is an SPD Bus Architecture (SBA) instruction used to send a message to an IOP (or AS/400 in this case) residing on the SPD bus.
4. The IOP places the Start Unit Operation (SUOP) message on the Bus for AS/400 to accept.
5. The Bus Control Adapter (BCA) hardware sends a Start Unit Operation Complete (SUOPC) message to the IOP to indicate that the Start Unit Operation message was received.
6. The Bus Control Adapter (BCA) hardware places a Received Unit Message (RUM) on the IO Event Stack on AS/400. A Received Unit Message is a entry written to the I/O event stack when an IOP sends a unit message to AS/400. The Received Unit Message contains the IOP address and the Opstart message from the IOP.
7. The Bus Control Adapter (BCA) hardware interrupts the AS/400 processor with an external interrupt to inform the AS/400 that there is a message to process.
8. The SLIC external interrupt handler handles the interrupt and calls a Bus Transport Mechanism method to handle the opstart message. SLIC Physical I/O notifies the Bus Control Adapter (BCA) hardware that the interrupt has been handled.
9. The SLIC Bus Transport code informs SLIC IPCF that an inbound exchange has been received.
10. SLIC IPCF calls a BusTransport method to check the RRCB for errors and initialize information needed by BusTransport to access the descriptors in the RRCB.
11. The SLIC IPCF code informs the IPCF client (I/O Hardware Driver) that a request has been received from the IOP.
12. The IPCF client processes the request and sends a response to the request. See section 3.2.3.7.3.7 “Sending a Response to a Request Sent by IOP” on page 3-92 for additional information.

3.2.3.7.3.7. Sending a Response to a Request Sent by IOP

The following processing is done to send a response to a request that was received from an IOP:

1. The IPCF client (I/O Hardware driver) processes the request. The IPCF client uses IPCF client interfaces to get the request and data from the inbound exchange. The IPCF code calls Bus Transport Mechanism methods that get the request and data from the Request Response Control Block.
2. If the IPCF client would like to send response data with the response, the IPCF client uses an IPCF client interface to add response data to the inbound exchange. The IPCF code calls a Bus Transport Mechanism method that adds the response data to the Request Response Control Block.
3. The IPCF client (I/O Hardware driver) uses an IPCF client interface to send a response to the exchange. IPCF calls a Bus Transport Mechanism method to send the response to the exchange. Bus Transport builds or sends an OPEND message. An OPEND message indicates to the requestor (IOP) that the server no longer needs access to the data areas specified in the RRCB and may also carry completion status information.
4. The Physical I/O Layer is responsible for sending messages on the SPD bus. The Physical I/O Layer builds a Start Unit Operation (SUOP) message. The Start Unit Operation message is an SPD Bus Architecture (SBA) instruction used to send a message to an IOP residing on the SPD Bus.
5. The Physical I/O Layer sends the Start Unit Operation message. The Bus Control Adapter (BCA) hardware places the message on the bus for the IOP to accept.
6. The IOP takes the message off the bus and acknowledges back to the Bus Control Adapter (BCA) hardware to indicate that it has received the message.
7. The Bus Control Adapter (BCA) hardware places a Start Unit Operation Complete (SUOPC) on the IO Event Stack (IES) with the bits stating the SUOP was successful.
8. The Bus Control Adapter (BCA) hardware interrupts the AS/400 processor with an external interrupt to inform the AS/400 that there is a message to be processed.
9. The SLIC external interrupt handler handles the interrupt and notifies SLIC BusTransport that the interrupt has been received. SLIC BusTransport processes the interrupt and matches up the SUOPC with the outstanding SUOP. SLIC Physical I/O notifies the Bus Control Adapter (BCA) hardware that the interrupt has been handled.
10. SLIC BusTransport informs IPCF that the response has been sent to the IOP.
11. The IPCF client resets the inbound exchange to indicate that it can be used for another incoming request.

3.2.3.7.4. I/O for Workstations and Printers

This section describes input and output (I/O) for workstations and for printers. Figure 3-23. “Detailed View of I/O for Workstations and Printers” on page 3-93 expands upon Figure 3-19. “High-level View of I/O” on page 3-80. In addition, the figure shows the layers of the SNA used by the AS/400 to provide I/O at both the OS/400 and SLIC layers. Everything above the MI interface shown in the figure is discussed in section 4.1.3.9 “OS/400 I/O Management” on page 4-59. The sections that follow discuss the SLIC layer, that is, the source/sink component, which provides the MI objects for I/O and the tasks that implement the MI instructions operating on these MI objects; the device IOMs and the line IOMs that form the bridge between the Source/Sink MI instructions and the IPCF, which transports messages from the IOMs to the BTM. The paragraphs below describe the layers shown in the figure in more detail.

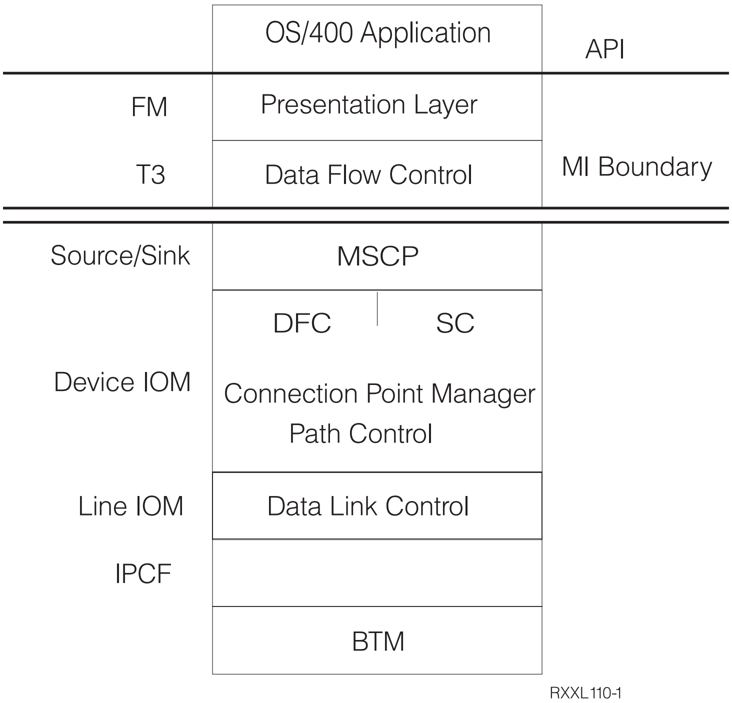


Figure 3-23. Detailed View of I/O for Workstations and Printers

3.2.3.7.4.1. Source/Sink Component

The source/sink instructions (SI) component provides a set of machine objects and a set of tasks that implement source/sink MI instructions to operate on those MI objects.[35] Only one of the set of source/sink MI objects, the logical unit description (LUD) is discussed here.[36] LUDs represent devices below the MI. All I/O is directed via the logical template of the physical device provided by the LUD.

SI owns all interfaces to the LUD. These interfaces include create, destroy, materialize, and modify MI instructions for each source/sink machine object type, including the LUD. LUDs are present for some devices in the base configuration. Other LUDs are created as desired. The LUD must exist before the new hardware device can be used. The MI instruction Request I/O (REQIO) is used to cause I/O to happen.[37] The MI instructions Request I/O and Modify Logical Unit Description are used to control I/O devices.

From the SNA viewpoint, the machine service control point (MSCP) is the part of the SI that provides routing and other necessary SNA services. This SNA layer implements tasks to coordinate the processing of REQIO requests and to support the Modify MI instructions for sink/source objects. MSCP, as part of the SI, does this by converting an MI instruction into a Send/Receive Message (SRM), which it places on the main

[35] The source/sink management component (SS) provides SLIC tasks for management services for source/sink.

[36] The controller description (CD) is functionally equivalent to the LUD. For example, a CD can describe a workstation controller. The other source/sink MI objects are not in the evaluated configuration. The CD and the LUD are described in section 3.2.1.5 “MI Objects” on page 3-26.

[37] The Request Path Operation MI instruction, which directs I/O to a Dump Space machine object (and indirectly causes DASD I/O), is included in the discussion of the Request I/O MI instruction.

input queue of the target IOM. All control of a LUD is via SRMs received by the main input queue of the IOM. These messages can be of four types:

- Modify LUD messages[38]
- Request I/O messages
- Load/Dump messages
- IPCF response messages.

The Modify LUD and Request I/O messages cause the IOMs to take action, while the Load/Dump and IPCF response messages are in response to the first two kinds. Load/Dump is discussed further in section 4.3.2 “Backup and Recovery” on page 4-90 IPCF response messages are discussed in section 3.2.3.7.1 “Inter-Process Communication Facility (IPCF)” on page 3-80.

3.2.3.7.4.2. Device IOMs

Device IOMs, like other IOMs, have a common structure. Each IOM has a module to handle Modify Logical Unit Description instructions, a module to handle Request I/O instructions, a module to handle Load/Dump instructions if the device can be used for Load/Dump, and a module to handle IPCF responses. In addition, an IOM has a module for SLIC common error recovery, and a module for the debug control function.

Each device IOM is created as a SLIC task when the device is varied on. Each of the source/sink objects contains a control state field. This field shows whether the device represented by the object is varied on or off. As part of the processing when a device is varied on, the SI uses the resource configuration record built at IPL time to verify that the LUD corresponds to a physical device installed on the system. This record is accessed only during vary-on processing.

One IOM is created per device. The MSCP finds the name of the IOM’s *PGM object in the LUD, dispatches the IOM task to the TDQ, and sends the Modify LUD SRM with the vary on parameter to the input queue of the IOM.[39]

The device IOM for workstations and printers is a SLIC task that implements the SNA path control, session control, and part of the data flow control layers for devices attached to the AS/400. Device IOMs for tape and diskette devices, and for DASD, are described in the sections below.

3.2.3.7.4.3. Line IOMs

For devices attached to a non-programmable workstation controller, the line IOM receives output data from the device IOM or passes input data to the device IOM, depending on the direction of the I/O request. The line IOMs implement the SNA data link control layer. For devices using the SNA model, the line IOMs add SNA headers to data being sent to the IOP for each device. See section 3.1.1.2.2 “Input/Output Processors” on page 3-12 for further details.

3.2.3.7.5. I/O for Tape, Optical, and Diskette Devices

As can be seen from Figure 3-24. “Detailed View of I/O for Tape, Optical and Diskette Devices” on page 3-95, I/O for tape, optical, and diskette devices is similar to I/O for workstations and printers. The Function Manager (FM) for a tape device is TA, for a diskette device is DK, and for a optical device is MO. These FMs format the Request I/O MI instructions directly, instead of depending on a T3 component to do this. Below the MI boundary, the source/sink component relays the I/O request to the proper device IOM. There are no Line IOMs; instead, the device IOM formats the IPCF command and calls the IPCF directly.

[38] These include supervisory service requests, such as reloading microcode to an IOP.

[39] In addition, the Modify LUD SRM with the “vary on” parameter causes the IPCF to open a functional connection to the IOP for that device. A Modify LUD SRM with the vary off parameter closes the functional connection. See above for more about the IPCF.

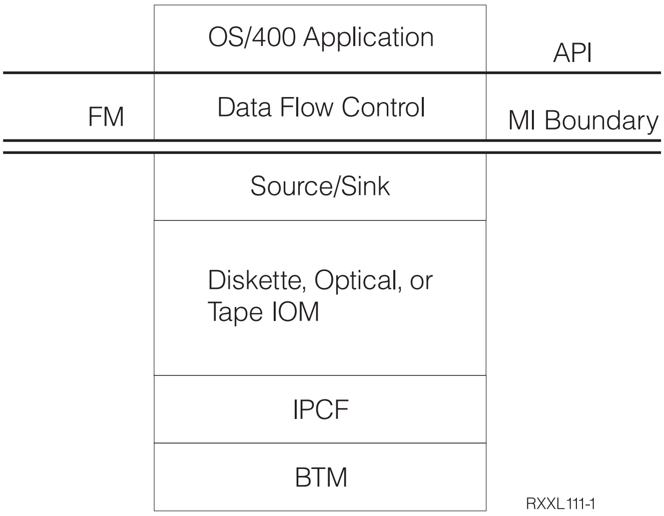


Figure 3-24. Detailed View of I/O for Tape, Optical and Diskette Devices

3.2.3.7.6. I/O for DASD

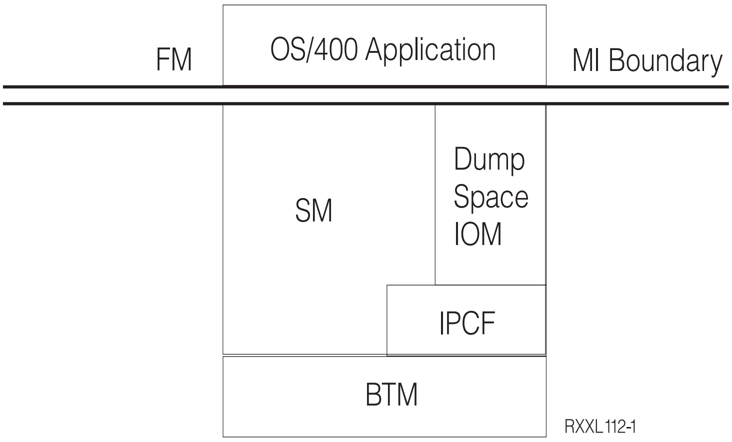


Figure 3-25. Detailed View of I/O for DASD

Figure 3-25. “Detailed View of I/O for DASD” on page 3-95 shows how the SM component is integrated into the I/O process. (The SM component is described in section 3.2.3.3 “Storage Management” on page 3-47) The dump space IOM, used by the save/restore component, also does DASD I/O. The dump space IOM uses the IPCF to send messages to the IOPs. The SM component uses the IPCF, or, in some cases, formats its own messages (using IPCF formats) to send to the IOP; this is done for historical reasons.

3.2.3.7.7. I/O for Cryptographic Processor

The cryptographic I/O is much like communications, as depicted in Figure 3-26. “Detailed View of I/O for Cryptographic Processor” on page 3-96, except that the only layer above the MI is the MI instruction CIPHER. The MI instruction is blocked; thus, UDSS APIs are provided and the APIs become the applica-

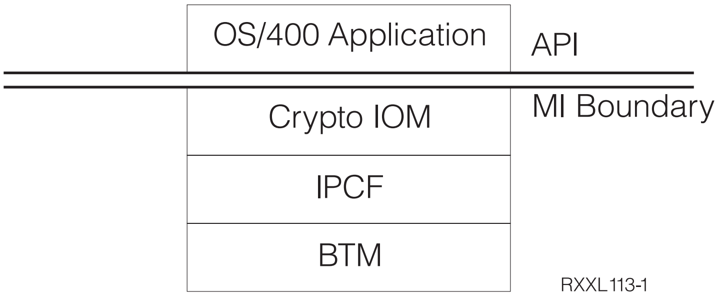


Figure 3-26. Detailed View of I/O for Cryptographic Processor

tion layer interface to the cryptographic processor. Below the MI layer is an IOM with which the CIPHER MI interacts. The MI instruction passes a space pointer to the input area and output area. The encrypted data is returned to the output space and the input space remains unaltered. The IOM pins the storage pointed to by the two space pointers for DMA usage by the crypto IOP. The IOM runs as part of the user code but in system state and uses system domain storage and a system domain activation group. There are no line IOMs; instead, the device IOM formats the message (see section 3.2.3.7.1 “Inter-Process Communication Facility (IPCF)” on page 3-80 for a complete description of IPCF functions) and transports the IPCF message directly to the BTM, then to the IOP. The IOP is varied on-line via a a CL command; if the IOP is not varied on-line then the MI instruction returns an error indicating this condition. The cryptographic processor uses the ANSI Data Encryption Algorithm (DEA) for encryption and decryption and a 64-bit key.

The cryptographic processor uses the Rivest Shamir Adleman (RSA) Public-key algorithm for key distribution and for generation and verification of digital signatures. Public-private key pairs used for digital signatures can be up to 1024 bits long, while key pairs for key distribution are limited to 512 bits. The export version of the cryptographic processor uses Commercial Data Masking Facility (CDMF) for encryption and decryption and a 40 bit key.

3.2.3.8. SLIC Exception Management

An exception represents an unusual condition within the system. The unusual condition may be detected by hardware or by software. Unusual conditions detected by the hardware cause hardware interrupts, which allow low-level interrupt handlers to try to resolve the condition in a transparent manner. If the interrupt handler cannot resolve the unusual condition, a SLIC exception is signalled. Unusual conditions detected by SLIC software also result in SLIC exceptions. Unlike unusual conditions detected by hardware, SLIC software can signal the exception directly.

SLIC exception management relates to the signaling of, monitoring for, and handling of exceptions at the SLIC level. Each SLIC component includes an exception handler for the exceptions that it expects to encounter.

Exceptions occur within the context of a task or process and they are related to the task or process in which they are signalled. The general framework of exception management also exists at the MI level, where errors can be user or program-defined. However, at the SLIC level, user or program-defined errors are not known.

Events differ from exceptions in that they relate to and define activities that occur during machine operation that may be of interest to users of the machine These are discussed in section 3.2.3.9 “SLIC Event Management” on page 3-100. In general, exceptions are synchronous and are caused by the execution of an instruction, whereas events are asynchronous and are caused by actions outside the currently executing instruction. From the user’s point of view, exception management is accomplished by the use of three exception message types (status, notify, or escape). See section 4.1.3.7 “Message Handling” on page 4-55.

Exceptions are not always an indication that an unrecoverable error has occurred. For example, “record not found” is an exception that a user can monitor for and handle. Exception management is a part of the normal flow of SLIC task execution and also of the layers above SLIC. The flow can be divided into two major sections, SLIC exception handling and MI exception handling.

3.2.3.8.1. Hardware Interrupt Handling by SLIC

During execution, the hardware may detect an unusual condition that requires immediate attention. Each interrupt type has an associated interrupt handler, which tries to handle the interrupt transparently if it can. The hardware generates a hardware interrupt, which causes control to be transferred to an interrupt handler. The interrupt handler to use is determined by the type of interrupt that is signalled. Examples of unusual conditions that cause interrupts are as follows:

- Page faults on data or instructions
- Illegal usage of an instruction
- Violations of security mechanisms on storage

At the time of the interrupt, the hardware saves some of the state of the machine in special purpose registers. The interrupt handlers are responsible for saving any additional machine state that might be changed during their execution. Many times, an interrupt can be handled transparently by the interrupt handler. If so, then the interrupt handler restores whatever state was saved and retries the interrupted RISC instruction.

If the interrupt handler cannot handle the interrupt transparently, a SLIC exception may be signalled. In general, exceptions are synchronously caused by the execution of an instruction. Because an exception is the direct result of an instruction executing in the current task (as opposed to some external event), the resolution of the exception is under the control of the current task and it must be handled by the current task before it can continue.

The signalling of a SLIC exception is done by invoking an Exception Management routine. This routine (known as “DoThrow”) is responsible for finding and invoking all SLIC level exception handlers that were enabled at the point that the exception was signalled. SLIC supports two general classes of exception handlers.

The first type of exception handler is known as a Retry/Continue exception handler. Retry/Continue handlers are generally designed to process exceptions that occur as the result of a hardware interrupt. A Retry/Continue handler has the ability to correct the condition that caused the error and to retry the failing RISC instruction, to continue execution at the next RISC instruction, or to defer to the next exception handler.

Retry/Continue handlers are represented by Component Specific Exception Handler (CSEH) blocks. CSEH blocks consist of the following:

- Address of the CSEH routine
- Address of the next CSEH block on the chain
- Definer’s invocation work area (IWA) stack pointer
- Optional component data

Retry/Continue handlers are run in LIFO order and they are managed by a TDE chain and by a chain on the stack invocation of the code that enabled the handler. The TDE chain is used to find and invoke Retry/Continue handlers. (For more information about the TDE, see section 3.2.3.1 “Tasks and Task Dispatching” on page 3-43.) The chain on the stack invocation that enabled the handler is used to manage the storage for the CSEH block. CSEH blocks usually exist within stack storage, but they can sometimes be allocated from heap.

The second type of exception handler is known as a Cleanup handler. Cleanup handlers are generally designed to handle any type of SLIC exception that may be signalled, including exceptions that occur as the result of a hardware interrupt not being handled by a Retry/Continue handler. Cleanup handlers generally perform some resource deallocation and error logging. They may specify an arbitrary piece of code to resume execution at or they may defer to the next exception handler. Cleanup handlers are run in LIFO order.

There are two different implementations of Cleanup handlers, each of which is defined by the source language that is used to enable the handler. In one implementation, Cleanup handlers are represented by CSEH blocks managed by a chain on the stack invocation that enabled the handler.

In the other implementation, active exception handlers are tracked by a finite state machine. The finite state machine is composed of an index in the stack invocation for the code that enables the exception handler and a statically generated table. The index is adjusted each time an exception handler is enabled or disabled. When an exception is signalled, the index and the table may be examined to determine which exception handlers need to be invoked.

DoThrow gives Retry/Continue handlers the first opportunity to handle SLIC exceptions. Unless specified otherwise, all possible Retry/Continue handlers are invoked and then DoThrow invokes all possible Cleanup handlers. A mechanism to explicitly order the way handlers are found and invoked is provided, making it possible to intersperse Retry/Continue handlers and Cleanup handlers in special situations. If the exception is not handled by a Retry/Continue handler or by a Cleanup handler, the exception is passed to the MI exception router for handling at the MI level. Before the MI exception router is invoked, the SLIC exception is converted to an MI exception. This is necessary because MI programs monitor for MI exceptions, not for SLIC exceptions.

In the case of exceptions in SLIC tasks that have no MI invocations, all exceptions must be handled by a SLIC exception handler. If the exception is not handled and all SLIC exception handlers are exhausted, the task will be destroyed.

3.2.3.8.2. MI Exception Handling

The MI exception router presents MI exceptions to the MI by locating an exception description enabled by a program executing in the target invocation (OS/400 or user program).

The MI exception router uses different methods to determine enabled handlers for OPM and NPM.

3.2.3.8.2.1. OPM Exception Handling

OPM MI programs declare exception descriptions. Exception descriptions are program objects defined in the program Object Definition Table (ODT) (see section 3.2.3.10 “MI Translator” on page 3-101). Exception descriptions define the set of exceptions to be monitored (either all exceptions, a specific exception, or a class of exceptions), and specify an action to be performed when one of the defined exceptions occur.

Possible actions are as follows:

- Do not handle: (1) Ignore the exception and continue instruction execution, (2) Continue the search for another exception description monitoring the exception, or (3) Resignal to the immediately preceding invocation.
- Defer handling by recording the occurrence for later processing.
- Handle immediately: (1) Branch to the specific instruction, (2) Invoke an internal subinvocation, or (3) Invoke a program.

The locating of a handler implies to SLIC that the exception will be handled. That is, SLIC assumes the action taken by the located exception handler will either resolve the unexpected condition in a manner that normal processing can proceed or signal a new exception because it failed to resolve the unexpected condition.

If an enabled exception description does not exist for the exception that has occurred, the MI process default handler is invoked (see section 4.1.3.7 “Message Handling” on page 4-55 for details).

3.2.3.8.2.2. NPM Exception Handling

The NPM provides the capability through the ILE language compilers to define multiple handlers, which are enabled by SLIC at execution. Exception descriptions define the class[40] of exceptions to be monitored. This is different from OPM, in which the user monitors on the basis of exception ID. Similar exception IDs are mapped to the same class, and the user monitors on the basis of the class. SLIC code maps the exception ID to the correct exception class.

Possible actions are as follows:

- Do not handle: (1) Allow the next handler within the same invocation to attempt to handle, or (2) Resignal to the immediately preceding invocation (percolation).
- Handle immediately by invoking an internal or external procedure.

An NPM handler may choose not to handle the exception and to allow the next handler within the same program to attempt to handle it. This is different from OPM, in which only one handler within a program is given a chance to handle the exception. If an enabled exception description does not exist for the exception that has occurred in an NPM program or if no enabled handler marked the exception handled in an NPM program, the MI process default handler is invoked (see section 4.1.3.7 “Message Handling” on page 4-55 for details).

3.2.3.8.2.3. Exception Handling and the State of the Process

Exception management may alter execution flow. The SLIC EX component ensures that the correct process state is restored whenever execution flow is altered (for information about the process state, see section 3.2.3.4 “Process Management” on page 3-61). The four cases are as follows:

- Branch point exception handlers (OPM)
- Internal exception handlers (OPM)
- External exception handlers (OPM and NPM)
- Modify resume point (OPM and NPM)

Exception management restores the state to the correct value before restarting execution in an invocation (program). This handles internal exceptions, branch point exceptions, and modify resume points.

External exception handlers are invoked with the process in the state of the defining invocation. As a result, user-state programs cannot cause the invocation of system-domain programs by causing an exception. The call to the handler tests state/domain compatibility and signals an error if a user-state program attempts to invoke a system-domain handler.

For all programs in the execution flow (stack) at any given time, the state of execution for the program is stored in internal structures. These structures are stored in the invocation control block (ICB) for OPM programs and in the activation control block (ACB) for NPM programs.

OPM programs allow processing to resume at the next MI instruction after exception handling. Since the pointer verification could have caused an exception, any pointers used on that MI instruction are viewed as requiring verification before next use. This ensures that pointers are not assumed to have been verified during a previous load into a register when exception processing could have bypassed that register load.

[40] The exception class is a 32-bit mask in which each bit represents a general type of exception condition. Examples of exception classes are authorization violation, domain/state protection violation, and object not found.

Exception processing for NPM does not allow a “resume at next MI instruction” option. Resumption is at the next RISC instruction. If an error occurs during a pointer verification instruction, SLIC exception management code gets control. This code ensures that the registers involved in the instruction are set to a value that produces an error if the executing code attempts to use the register (with the same contents) in a later operation. This prevents users from handling an error on the pointer operation and then attempting to use the results of the failed operation as valid.

3.2.3.8.2.4. System Calls

The RISC instruction System Call Vectored (SCV) provides the capability for a program running in problem state to request operations that must be run in privileged instruction state (i.e., a system call). SCV uses an SCV table which contains 128 different entry points to SLIC routines. SCV is invoked with an index, which causes a branch to one of the 128 different entry points. For a discussion about how the translator uses SCV, see section 3.2.3.10.6 “The Router” on page 3-105.

During execution of the RISC instruction SCV, the hardware and the SCV routine being branched to cooperate to save the state of the machine. The state is saved in registers by the hardware, which the SCV routine may then store in a new stack frame. The state must be restored to these registers before the SCV routine returns. The SCV routine uses the RISC instruction Return From SCV (RFSCV) to return to its invoker. After the RISC instruction RFSCV, the machine will be in its original state before the SCV was invoked.

3.2.3.9. SLIC Event Management

Events relate to and define activities that occur during machine operation that may be of interest to users of the machine. Event management enables an MI process to monitor these events and perform specific functions based on their occurrence. The events being monitored can represent conditions either internal or external to the monitoring process, that is, one process can be monitoring conditions caused by the same process, by other processes that are currently in the machine, or by some condition not directly related to the existence of any process in the machine.

Machine events are signalled as a result of certain conditions detected by the machine. Explicitly signalled events can be defined (based on user protocol), signalled, and monitored through the event management instructions. Events can be signalled to event monitors in all processes or to a specific process. Generally, a process monitors an event so that some function, which is related to the conditions that caused the event, can be performed.

There are two major classifications of machine events, object-related and machine-related.

Object-related events are conditions directly related to the values and the attributes of an object. For example, the message limit is reached on a queue, or an object authority failure occurs.

Machine-related events are conditions that are not directly related to a system object. For example, a specific timer interval has elapsed or a machine error has occurred.

In addition, a group of events have no defined meaning to the machine because they are defined outside the machine range by the MI user. These events are based on user protocol. The same event management facilities available for machine events are also available for MI user defined events and, therefore, may be used for intraprocess monitoring and interprocess communication and synchronization.

Events are identified by the following components:

- Event class - The class of events is generally related to a specific type of object, a machine function, or a condition. For example:
  - Queue events
  - Authorization events
  - Machine status events
- Event type - Type is the subclassification of the event within a class to further identify the event. For example, types are defined for the class of machine status events: the machine check, machine power, and error log full.
- Event subtype - Subtype is a further subclassification of the event within the type. For example, within the machine power type, the subtype specifies the machine power status such as a power controller failure.

A specific event can be monitored by specifying the entire event identification (class, type, and subtype). Events can also be monitored generically. Generic monitoring can be by class (the occurrence of any event within the class without regard to type or subtype) or by type (the occurrence of any event within a particular class and type without regard to subtype).

Event monitoring can be further limited by specifying a qualifier. The qualifier is termed a *compare value* and can be a reference to a system object (through a system pointer) or a scalar.

Events related to system objects can be monitored for all occurrences of an event (for example, the message limit reached on any queue). Events can also be qualified (by specifying the appropriate event monitor compare value) such that monitoring occurs only if the event occurs for a specific system object (for example, the message limit is reached on queue XYZ).

Events that have associated compare values can be monitored so that their occurrence is detected only if a specified compare value matches the condition (compare value) in the event. For example, timer events are signalled if the time interval specified in a compare value has elapsed or if the time of day specified in a compare value has been reached.

3.2.3.10. MI Translator

On AS/400, only bit strings forming RISC instructions in an encapsulated MI program object can be executed.[41] To execute code on AS/400, a program must be defined as an MI template, translated to RISC code, and encapsulated in an MI program object.

An MI program object is the encapsulated and executable form of a program template defined at the MI and used as an input parameter to the MI translator. The resulting program object logically contains both the functional definition and the data definitions from the MI program template. The subsequent execution of a program causes a series of functions to be performed against a set of MI objects.

The MI has two variations that share many characteristics but take different approaches: the original MI (OMI) and the new MI (NMI). The process of translation consists of compiling a program defined in terms of MI instructions into an encapsulated RISC executable program that performs the functions as defined at the MI. The SLIC contains a single translator, which recognizes only NMI instructions. A NMI instruction stream is produced by the combined efforts of the ILE compilers, the XPF CUBE component, and by the MI

[41] External users can only create RISC instruction streams and cause them to be executed via MI program objects. SLIC instruction streams, invoked by the MI, are not contained in MI program objects and the external user has no ability to create SLIC instruction streams. MI cursor objects contain small RISC instruction streams created by the SLIC database component but the external user has no ability to directly create these instruction streams or to directly invoke them.

transformer.[42] The MI transformer converts OMI instruction streams to NMI instruction streams for consumption by the MI translator. The following sections discuss translation of OMI and NMI program templates.

3.2.3.10.1. OMI Instructions

The OMI program template describes a program to be created. The template header defines the set of attributes common to all system objects as well as information unique to programs. Three components are found in the template: the OMI instruction stream, the object definition table (ODT), and user data. Each component is optional. If present, its location is indicated by offset values specified in the template header.

The OMI instruction stream defines the set of operations to be performed by the program. It is composed of a series of instructions in an operator-operand format.

An instruction can be viewed as a vector of entities, where each unit can be one of the following fields:

- Instruction operation code field (required)
- Operation code extender field (optional)
- Operand field (zero to four primary operands/zero or more secondary, branch, or indicator operands)

The object definition table (ODT) is the means for defining all program data that is referenced by the operands in the instruction stream.

The remaining optional component of a program template is user data. The user defines the format and contents of the user data area, which are placed in the encapsulated part of the object. The SLIC then maintains this area as part of the program. The associated space of *PGM objects is used by OS/400 to maintain information about the program. The encapsulated program object is protected by read-only from either user or system state hardware storage protection.[43] The associated space has hardware-storage protection of read-only from user state.

3.2.3.10.2. OMI Program Translation

The MI instruction Create Program (CRTPG) converts an MI program template into an encapsulated MI program. The CRTPG instruction causes the invocation of the MI transformer. Input to the MI transformer is the program template; output from the transformer is a set of NMI templates. The MI transformer converts the OMI instruction stream and ODT components found in the program template to an equivalent NMI representation. The NMI representation of the program template is passed by the transformer to the MI translator to create an MI module object. The MI transformer then passes the resulting MI module object to the program binder to create an encapsulated MI program object.

3.2.3.10.3. NMI Instructions

In addition to OPM programs, which are described by an OPM program template, the NPM supports a new program template syntax for MI, called new MI (NMI). The NMI template is used to create an MI module object, which cannot be run until created into a program by the binder, as explained in section 3.2.3.10.5 “NMI Program Binder” on page 3-104. An NMI module object is represented in the NMI template as a number of template extensions called components. Each component consists of a header that describes the

[42] The MI transformer is a SLIC routine that takes an OPM program template (an operand of the MI instruction Create Program) and transforms it into an equivalent NMI program template while maintaining the semantics of the OMI program. This allows a single MI translator to generate RISC instruction streams for both program models.

[43] The first page of the encapsulated program object is only protected with read-only from user state hardware protection because the system needs to update information contained in the first page of the encapsulated object.

component, including its length and sizes of any work areas, followed by a table or structure that contains some information. The NMI module object templates include the following components:

- Dictionary Component
- Instructions Component
- Binding specifications
- Initialization Component
- Alias Component
- Type Information Component
- Literal Pool Component
- String Directory Component
- Copyright Component

The dictionary component contains the dictionary definition table. The dictionary definition table describes all the data objects defined or referred to in the compile unit, plus all procedures, exception handlers, literals, and label constants. The instructions component contains the instruction stream, which defines the executable code of the compile unit. The code is packaged into procedures that follow one after another in the instruction stream.

The instructions component contains procedures, which are callable units. A procedure can be any one of the following types, where the type defines the scope of the procedure:

- Exported. This type of procedure is exposed to other compile units and can be called from outside the module.
- Imported. This is a reference to a procedure defined in another compile unit.
- Internal. This type of procedure is not exposed outside the compile unit.
- Program entry. When bound into a program object, this is an entry that can be given control when the program is invoked. A module can contain at most one procedure of this type. Each module in a program object may have one procedure of this type, but only one module can export a procedure of this type.
- Based. This type of procedure can be used to describe generic dictionary entry functions called through a pointer

OMI instructions are storage-to-storage instructions that use a complex instruction set. OMI programs are dynamically bound at runtime. By contrast, NMI instructions are stack oriented. NMI programs can use dynamic runtime binding or can be bound prior to execution. While the standard instructions for NMI (e.g., add, branch) are different from OMI instructions, the complex instructions that operate on objects are the same.

The binding specifications contain binding-related data to be used when the module is bound into a program. The initialization component contains information used to initialize static data objects defined in the compile unit. The alias component is optional and, when provided, contains the alias classes and alias unions defined by the language compilers. The type information component is optional and, when provided, contains data type and length information that can be referred to from special load and store operations. The literal pool component is optional and, when provided, contains the values of string literals. The string directory component contains the names that are input to the CRTMOD instruction. One of the chief uses of the string directory components is to contain the names of symbols referred to by the binding specification entries. The copyright component allows the encapsulation of copyright statements into a module object and into any program created from the module.

3.2.3.10.4. NMI Module Object Translation

The MI instruction Create Module (CRTMOD) converts NMI templates into an encapsulated MI module object that contains RISC instructions. The CRTMOD instruction causes the invocation of the MI translator. Input to the translator is the NMI module object templates; output from the translator is a set of RISC instruction streams and supporting structures. The Module Object Manager (MM) component packages the instruction streams and structures into an MI module object. The module object cannot be invoked until the MI instruction Create Bound Program (CRTBPGM) is run, which causes the binder to bind the module to zero or more other modules. CRTBPGM resolves references between modules and creates an encapsulated MI program object that is executable. Both the CRTMOD and the CRTBPGM instructions are blocked.

The overall design of the translator is similar to that of a compiler. The basic difference between a compiler and the translator is that the translator's input is bits rather than character strings and the first error encountered by the translator, such as a syntax error, a blocked instruction, or an exception, stops the translation process. The translator is divided into several major phases, which start with processing the dictionary component that comes into the translator and concludes with the final code emission phase.

The driver component of the translator is responsible for initialization and control of the translator phases. The driver component enables the component-specific exception handler and allocates and initializes a common area that is used by all the translator phases to communicate. Information retrieved from the NMI template, such as the optimization level, is saved in this common area.

The first phase scans the dictionary component and determines the storage allocation data objects will have at run time. If an instruction stream is present, the NMI is rewritten in the next phase to an internal representation. The rewrite phase is when the NMI opcodes are examined to ensure only valid unblocked MI opcodes exist in the template. The translator generates an exception if there are any syntax errors, including attempts to use a blocked instruction, and establishes internal areas for later phases of the translator. To determine whether an MI opcode is blocked or unblocked, the NMI translator uses a table of MI opcodes. Subsequent phases involve several types of optimization. The result is a set of code improvements that are then applied to the internal representation. The internal form of the instruction stream is rewritten once more into another lower-level representation of the instruction stream that is more suitable for performing machine-dependent optimizations, such as register allocation and instruction scheduling. Instruction scheduling reorders the instruction stream to take advantage of pipeline and overlap characteristics of the hardware.

The final phase builds the actual RISC instruction stream. The instruction streams from each procedure are collected and are eventually encapsulated in a module object.

3.2.3.10.5. NMI Program Binder

An MI module is bound into an encapsulated MI program by executing the MI instruction Create Bound Program (CRTBPGM), which invokes the binder.

CRTBPGM uses a set of templates. The object list template and the symbol resolution list are the main templates that are input to the binder. The object list template contains system pointers to all bindable objects, and the symbol resolution list is used to connect an imported symbol to the exported symbol that satisfies the import.

Bindable objects are either module objects or service program objects.[44] Service programs represent a set of exported procedures and data that can be used to satisfy module imports, providing a function similar to a dynamic link library.

Using the import and export references, the binder resolves addressing to the various modules and supplies information for resolving to service programs at activation.

[44] A module object is bound by copy into a bound program object. A service program is bound by reference to a bound program object.

In addition, checks are made to prevent attempts to bind system-domain modules and service programs with user-domain programs.

If all checks pass, the modules being bound together are encapsulated in the program object and addressability is set between procedures bound within the program. When complete, an MI program object is created with the header set to user domain and user state.

3.2.3.10.6. The Router

For most of the complex object-manipulation MI instructions, the translator does not generate code inline, but generates RISC instruction SCV (call) to a SLIC runtime routine. The hardware provides only 128 entries in the system call vectored table (and the system supports more than 128 MI instructions), so the code sequences generated for most complex MI instructions all use the same entry in the SCV table to make an indirect call, through an MI router. The MI instruction parameters or their addresses are passed to the router, as is the routing code to the appropriate SLIC routine.

The MI router performs some common functions, then uses the routing code as an index into its own table of SLIC runtime routines. One of these common functions is a check for blocked instructions. A state flag is associated in the router's internal table with each runtime routine to which it can pass control. If the execution state is user and the system-only flag is on, which indicates a blocked instruction, an authorization exception is signaled. If the state test passes, the router then calls the routine for the indicated MI instruction. All blocked MI instructions are ones that are invoked with an SCV. The router takes care of the runtime blocked instruction check for all MI instructions that go through the router.

The code generated for a few of the complex instructions specifies individual SCV table entries. Such implementations then do not use the MI router and, if the MI instruction is blocked, perform a check against the current execution state to test whether the instruction is allowed to be executed.

Section 4
Software Architecture

4.1. OS/400

This chapter describes the objects that are accessible at the OS/400 level, their interfaces, and the functional areas and Licensed Products that comprise OS/400, as specified in Appendix B, “Evaluated Software”, on page B-1.

Figure 4-1. “The OS/400 Layer” on page 4-1 illustrates the functional components at the OS/400 layer. The major components of the OS/400 layer, expanded to the right side of the figure, are discussed in detail in this chapter. The objects that are managed in the OS/400 layer are discussed in section 4.1.1 “Object Management in OS/400” on page 4-2. The OS/400 interfaces are discussed in section 4.1.2 “OS/400 Interfaces” on page 4-14. The functions implemented in the OS/400 layer are discussed in section 4.1.3 “OS/400 Functional Areas” on page 4-18.

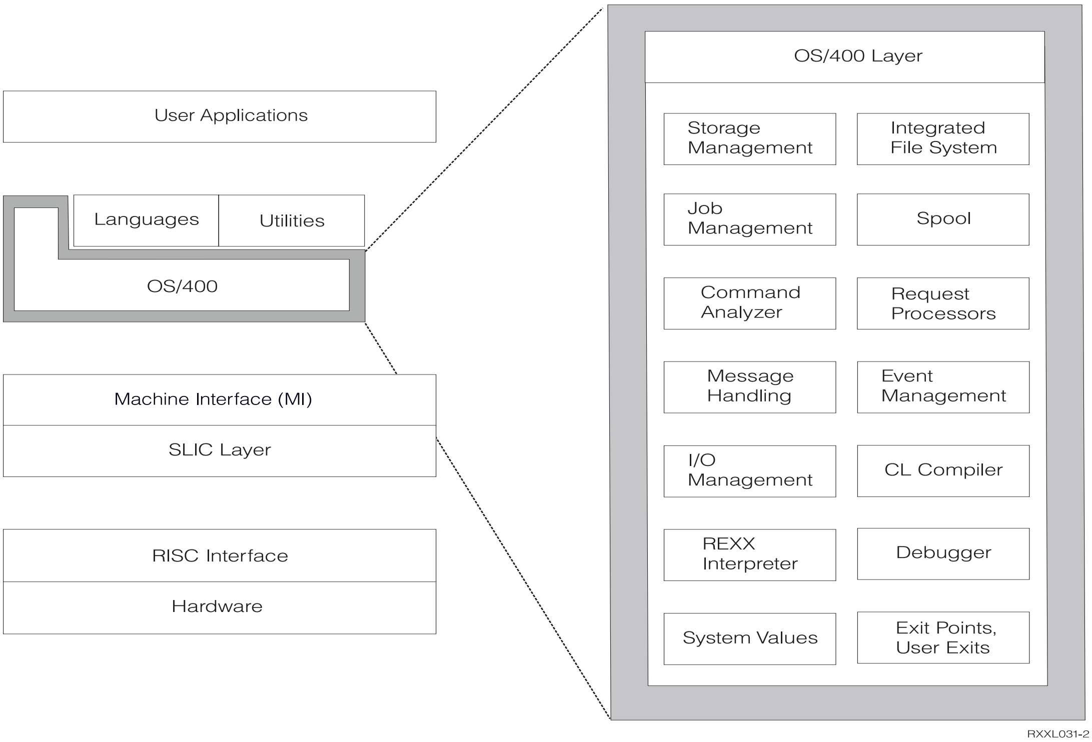


Figure 4-1. The OS/400 Layer

4.1.1. Object Management in OS/400

Above the Machine Interface (MI), the OS/400 operating system presents objects to the application programmer by grouping one or more MI objects together to form an *external object*. For example, a *FILE external object is built from four MI objects: a space, a data space, a data space index, and a cursor. When two or more MI objects constitute an external object, the object manager performs Discretionary Access Control (DAC) checks on only one of the MI objects. The MI object used for DAC checks is determined when the external object type is designed. For example, the MI space object is used for DAC checks on *FILE external objects.

Also above the MI level are *internal objects*, which are used only by OS/400 for system processing. This section describes the OS/400 objects that are constructed of one or more MI objects. These include external and internal objects.

4.1.1.1. External Objects

This section discusses management of external objects. The AS/400 external object types and their managing components are listed in Table 4-1 on page 4-3. Program management, data management, and library management are discussed in this section due to their complexity and importance.

Table 4-1 on page 4-3 lists each external object type, the managing component, the MI object from which the external type is built, and a brief description of the object type. External object types not included in the evaluated configuration are separated into two categories. These categories are discussed below.

The first category includes external objects that exist in the evaluated configuration, but are only accessible by a privileged user. Instantiations of these objects are created during installation or IPL, but commands that address these are restricted to trusted users. External objects that fall into this category are indicated in Table 4-1 on page 4-3 with a footnote.

The second category includes external objects that do not exist on a C2 System. These objects are not created by the system. The commands to create these objects are restricted to trusted users, and the Trusted Facility Manual (TFM) instructs these users not to create them. External objects that fall into this category are not included in Table 4-1 on page 4-3.

Each external object is an instantiation of an MI object (*FILE is built from multiple MI objects); thus, each external object has the segment and Encapsulated Program Architecture (EPA) header that all MI objects have.

OS/400 uses the associated space of the MI object to store information pertaining to the external object. For example, the user profile (*USRPRF) external object has an associated space. The associated space of the user profile object is used to store information pertaining to the user profile, such as the initial program, current library, and message queue for the user.

Many external objects are built from the MI space object. The MI space object has the segment and EPA header, and an associated space. An example of an external object that is built on an MI space object is the command (*CMD) object. The associated space of the *CMD object is used to store information pertaining to the command, such as the name of the command processing program (CPP), and a description of each command parameter, such as parameter name, parameter type, and parameter length.

The MI instruction Set Space Pointer from Pointer (SETSPPFP) is used to obtain addressability to the associated space of the external object. In addition to the authority required by SETSPPFP, the OS/400 program may require additional authority to the object in order to access the object.

The associated space of each external object type is protected in the same manner. The associated space of each system-domain object can be accessed only by a system state program since the associated space segment is also system domain. The associated space of the user-domain program object is protected via the hardware storage protection attribute of read-only from user state.

When an external object is created, the addressability to the object is inserted into the context (*LIB) object or data space index (*DIR) object. The owning user profile object is updated to reflect ownership of the object. These are the only three objects that are affected when an external object is created.

When an external object is deleted, the addressability to the object is removed from the *LIB or *DIR and from the authorization list (if an authorization list is securing the object). Also, references to the object are removed from the owning user profile and from all other profiles that have authorities to the object

Table 4-1. External Objects

Object Type	Managing Component	Based on MI Object	Description
*ALRTBL[a]	RU - Reference Code Table Utility	Index	Contains alert descriptions that define the contents of a Systems Network Architecture (SNA) alert for particular error conditions that are used to build an alert.
*AUTL	SY - Security	Authorization List	A list of users and their authorities to objects.
*BLKSF	P0L - Posix Function	Byte Stream File	Represents a User Defined File System.
*BNDDIR	BN - Program Binder	Space	Contains information used to combine modules into a program mapping.
*CLD	CX - C Runtime	Space	Contains formatting information for numeric and monetary quantities, collating sequences, and code point mapping.
*CLS	WC - Work Control	Space	Describes the run environment attributes for a job.
*CMD	CD - Command Definition	Space	Contains the description of a CL command.
*CNNL[a]	DC - Device Configuration	Connection List	Contains configuration information required by the system to manage how calls are sent or received across an integrated services digital network (ISDN).
*COSD[a]	DC - Device Configuration	Service Description	Contains ranges of link and node characteristics used to influence the routing of an APPC session through an APPN network.
*CRQD	FV - Systems management	Index	Describes the change request description object used by system management utilities.
*CTLD	DC - Device Configuration	Controller Description	Describes a system controller and its features.
*DEVD	DC - Device Configuration	Logical Unit Description	Describes the features of a device on the system.
*DIR	P0L - Posix	Data Space Index	Serves as the directory for stream file objects.

Table 4-1. External Objects

Object Type	Managing Component	Based on MI Object	Description
*DOC[a]	OD - Document Library Services	Space/Byte String Space	Contains the text of a document and descriptive information about it that has been filed in the document library.
*DTAARA	WC - Work Control	Space	Contains a data value that can be used and changed by different jobs.
*DTADCT	ID - IDDU	Space	Contains file, format, and field definitions used to describe database files.
*DTAQ	MH - Message Handler	Queue	Provides a method for sending data between jobs and field definitions used to describe database files.
*EDTD	EC - Edit code	Space	Describes an edit code mask used for formatting output fields.
*EXITRG[a]	US - User Object Support	Space	The object used to store registered exit points for AS/400 function.
*FCT	GS - AFP (Advanced Function Presentation) Print Resource Utility	Index	Contains a forms control table.
*FILE	DM - Data Management	Space, Data Space, Data Space Index, Cursor[b]	Contains, or provides access to, a group of related data records in the system including database, diskette, tape, printer, save, and display files.
*FLR[a]	OD - Document Library Services	Space	Contains documents and other folders. Serves as a directory for finding documents and folders by name.
*FNTRSC	GS - AFP (Advanced Function Presentation) Print Resource Utility	Space	An object representing how characters appear on a page.
*FNTTBL	PQ - PSF	Space	Allows users to create their own font and code page mapping tables that are searched before the internal (system supplied) mapping tables.
*FORMDF	GS - AFP (Advanced Function Presentation) Print Resource Utility	Space	An object containing medium mapping and copy control information.

Table 4-1. External Objects

Object Type	Managing Component	Based on MI Object	Description
*FTR	NM - Network Management Services	Index	A filter object allows the user to categorize data into groups and specify special actions to be applied to each group.
*GSS	GD - Graphics Display	Space	Contains character sets that can be used by a graphics application program to display characters other than the default character set supplied with the operating system.
*IPXD	DC - Device Configuration	Space	Describes the features of the Internetworking Packet Exchange Protocol Description (IPXD) device configuration object.
*JOBD	WD - SBS Description	Space	Contains a set of attributes used to control the running of a job.
*JOBQ	SP - Spool	Index	Contains entries for jobs to be processed by the system. The Spool component owns all interfaces to the *JOBQ external object, except SBMJOB, TFRJOB, and TFRBCHJOB. These three are owned by the Work Management (WM) component (see section 4.1.3.4 “Spool” on page 4-35 and section 4.1.3.3 “Job Management” on page 4-23).
*JOBSCD	WC - Work Control	Index	Contains a set of attributes used to control future submission of jobs based on date and time.
*JRN	JO - Journal	Journal Port	Contains information about journal receivers and database files recorded in a journal. It provides access to journal receivers.
*JRNRCV	JO - Journal	Journal Space	Contains journal entries created when database files are changed.
*LIB	LI - Librarian	Context/Index, Space	Serves as a directory for finding objects by name when they are to be used. Subtypes include production libraries and test libraries.
*LIND[a]	DC - Device Configuration	Network Description	Describes a communications line to the system.
*LOCALE	LG - NLS Support	Space	Locale is an object that contains cultural values, such as character classification, collating sequence, time formatting, and other cultural sensitive values. It is a binary object that can be created only with the locale compiler.
*MENU	UI - User Interface Manager (UIM)	Space	Identifies the text, messages, commands, and programs used to display a menu and process the user’s response (see section 4.1.3.6.1 “The User Interface Manager” on page 4-52).

Table 4-1. External Objects

Object Type	Managing Component	Based on MI Object	Description
*MODD[a]	DC - Device Configuration	Mode Description	A logical group of sessions with a common set of characteristics.
*MODULE	BN - Program Binder	Module	Object produced by a compiler that is used to create a program object (ILE environment only; see section 3.2.3.10 “MI Translator” on page 3-101).
*MSGF	MH - Message Handler	Index	Contains descriptions of predefined messages (see section 4.1.3.7 “Message Handling” on page 4-55).
*MSGQ	MH - Message Handler	Space	Contains messages being sent and received by the system and its users (see section 4.1.3.7 “Message Handling” on page 4-55).
*NTBD[a]	DC - Device Configuration	Space	Describes the features of the NetBIOS device configuration object.
*OUTQ	SP - Spool	Index	Contains entries for spooled output files to be written to an output device (see section 4.1.3.4 “Spool” on page 4-35 and section 4.1.3.3 “Job Management” on page 4-23).
*OVL	GS - AFP (Advanced Function Presentation) Print Resource Utility	Space	An object that represents electronic versions of preprinted information such as a letterhead.
*PAGDFN	GS - AFP (Advanced Function Presentation) Print Resource Utility	Space	Contains formatting controls for line data. It can include controls for the number of lines per logical page, font selection, print direction, and mapping of individual fields on the logical page.
*PAGSEG	GS - AFP (Advanced Function Presentation) Print Resource Utility	Space	An object representing a piece of a page. This includes standard paragraphs and signature information.
*PDG	CJ - PrintManager/400	Space	Space used to hold print descriptors, which contain user options for printing.
*PGM	BN - Program Binder	Program	Contains the code needed to perform the user’s task (e.g., CL and high-level language programs).
*PNLGRP	UI - User Interface Manager (UIM)	Space	Used to present information or help text on a display or printer device (see section 4.1.3.6.1 “The User Interface Manager” on page 4-52).

Table 4-1. External Objects

Object Type	Managing Component	Based on MI Object	Description
*PRDDFN	SZ - Software Management Services	Space	Contains product information that is loaded when a licensed program is installed on the system, such as the product name, product ID, copyrights, and modification level.
*PRDLOD	SZ - Software Management Services	Space	Defines a control object for a product option.
*PSFCFG	PQ - Print Server	Space	Describes features used for Advanced Function Printing.
*QMFORM	QX - Query Management	Space	Contains the format of the query or report that is defined by a user of the Query/400 Licensed Program and Systems Application Architecture (SAA).
*QMQRY	QX - Query Management	Space	Contains the data to be returned from a query or report that is defined by a user of the Query/400 Licensed Program and SAA.
*QRYDFN	QU - Query	Space	Contains information about a query or report that is defined by a user of the Query/400 Licensed Program.
*RCT	RU - Reference Code Table Utility	Index	Contains information used to help with problem analysis to detect hardware errors.
*SAVF	SR - Save Restore	Dump Space	Contains saved objects.
*SBSD	WD - SBS Description	Space	Describes a subsystem and its operating environment in the system.
*SCHIDX	UI - User Interface Manager (UIM)	Index	Contains information used by the index search function (see section 4.1.3.6.1 “The User Interface Manager” on page 4-52).
*SPADCT	DS - Dictionary Services	Dictionary	Contains a list of words, synonyms, and hyphenations.
*SQLPKG[a]	SQ - SQL	Program	Contains information needed by SQL to perform a request of a program on a remote system.
*SRVPGM	BN - Program Binder	Program	A service procedure called by a new MI (NMI) program (see section 3.2.3.10 “MI Translator” on page 3-101).
*STMF	P0L - Posix Function	Byte Stream File	Describes the stream file object.

Table 4-1. External Objects

Object Type	Managing Component	Based on MI Object	Description
*SYMLNK	P0L - Posix Function	Byte Stream File	Used to implement the symbolic link to a stream file object.
*TBL	TB - Table	Space	Contains a set of values used to define a byte-by-byte translation of data, or to define a collating sequence.
*USRIDX	US - User Object Support	Index	Provides search functions and automatically arranges data based on the value of data.
*USRPRF	SY - Security	User Profile	Identifies a user to the system and specifies what system resources and objects he or she can use.
*USRQ	US - User Object Support	Queue	Provides a means of asynchronous communication between two processes.
*USRSPC	US - User Object Support	Space	Contains a large amount of data that can be changed and used by a user.
*VLDL	SY - Security	Index	Contains a list of identifiers that can be validated.
*WSCST	UC - Workstation Customization	Space	Used to customize a workstation for a particular user, for example to change the keyboard mapping for a user.

[a] These external objects exist in the evaluated configuration, but are accessible only by a privileged user. Instantiations of these objects are created during installation or IPL, but commands that address these are restricted to trusted users. In addition, the underlying MI objects are created as system domain objects; they are not accessible to users via MI instructions. Because they are restricted, the MI objects, connection list, mode description, and service description, are not listed in Table 4-1 on page 4-3.

[b] Some *FILE objects are made up of multiple MI objects, and some are made up of only one.

4.1.1.1.1. Program Management

The *PGM comprises a program MI object. *PGM objects are created in user domain and execute in user state. However, most IBM-supplied *PGMs reside in system domain and execute in system state.

When a program is created, the creator can specify that the program is to have the adopt user profile attribute. Furthermore, a program with the adopt user profile attribute can have the propagate attribute. A program with these two attributes makes its adopted user profile authorities available to any additional called programs. Except for authority checks of directory (*DIR) and stream file (*STMF) objects, program adopted user profiles provide an additional source of authority for satisfying authorities needed by a process.

If the ownership of a program is transferred, the user profile of the new owner is used as the adopted user profile when the program is subsequently invoked. Ownership of a program with the adopt user profile attribute cannot be transferred unless the process transferring the ownership has all-object special authority. A currently invoked, transferred program that has the adopt user profile attribute continues to use the old owner's adopted user profile until a new invocation occurs.

The only way to create an executable program is through the translator (see section 3.2.3.10 "MI Translator" on page 3-101). For OMI programs, the translator, invoked through the MI instruction Create Program, converts the MI instructions contained in a program template into an executable RISC program. The resulting program dynamically links to other programs at execution time.

For NMI, the MI instruction Create Module produces a module object that contains RISC instructions, but cannot be invoked. The MI instruction Create Bound Program invokes the *BINDER* (linkage editor) to resolve references between the modules to form an executable *PGM.

4.1.1.1.2. Data Management

The Data Management (DM) component was initially provided to isolate the AS/400 system programmer from the concerns of device-specific programming. DM allows users' programs to open, interact, and close devices without having to know device-specific information.

DM manages the *FILE subtypes listed below. For the remainder of this section these subtypes will be referred to generically as *files*. All file open requests are routed to DM. DM performs part of the open, then passes control to the Database (DB) component or the Device File (DF) component, depending on the subtype, to finish the open. In addition, DM manages *overrides*. Overrides provide a way for the programmer to make changes (e.g., override the file name) without actually changing the program or file.

- Database
  - Physical
  - Logical
  - Source
- Device
  - Diskette
  - Display
  - Printer
  - Tape
  - Save

DM opens a file (database or device) after checking the user's authority (object operational and at least one data authority) to the file. During the execution of an open, DM copies the *Open Data Path (ODP)* and initializes it in the user's process space.

This section describes the open function performed by DM on both database and device files. To open database files, DB provides DM with file information (see section 4.2.1.6 "Database Management System Transaction Processing" on page 4-81), which is then registered in DM control blocks. For device files, once DM has verified DAC and obtained a lock on the I/O device, I/O support for the opened device file is performed by one of the DF Function Managers (FMs) (see section 4.1.3.9.1 "Function Management" on page 4-60).

The relevant data structures used by DM to perform file opens are described below. Following these descriptions, the open function is described.

4.1.1.1.2.1. Opened Data Path

The Open Data Path (ODP), which is hardware-storage protected, is a process-local control block for an opened file. The ODP contains a feedback portion where the open process and the FMs post I/O request completion status. The ODP also contains file information and open options. During the open process, the ODP is copied from the file. Its purpose is to provide a repository for all late-bound information about the file.

The ODP is created in two forms. When a file is created, an associated *prototype* ODP is also created. When a file is opened, an *active* ODP is created from the prototype ODP using the MI instruction Create Duplicate

Object. The active ODP is created in the process-local area, in user domain, and is write-storage protected. The active ODP is destroyed when the file is closed.

4.1.1.1.2.2. User File Control Block

The User File Control Block (UFCB) provides a linking mechanism to all objects that make up the data path. After the file open is executed, the UFCB contains status completion information and serves as a link from the program to the active ODP.

The UFCB can be created in two different ways, depending on whether the file is opened as a *simple open* or as a *query open* (see section 4.2.1.6 “Database Management System Transaction Processing” on page 4-81). If the UFCB results from the simple open, UFCB is stored in the automatic storage space of the process-local space. If UFCB creation results from the query open, UFCB is stored in temporary space. Addressability to the UFCB is stored in the DMCQ. The UFCB contains addressability to the ODP.

4.1.1.1.2.3. Data Management Communication Queue

The Data Management Communication Queue (DMCQ) keeps track of process-oriented files. For example, the DMCQ manages the overrides issued in a process, ensuring that they are applied appropriately when files are opened. It also facilitates cleaning up overrides when the job, invocation, or activation group to which they are scoped terminates, or when an override is explicitly deleted.

DMCQ also keeps track of all opened files within a process. In the case of abnormal process termination, the condition of opened files can be determined from the DMCQ, and a more graceful DM function conclusion is accomplished.

DMCQ (an MI space object) is created as part of the temporary job structure (see section 3.2.3.4 “Process Management” on page 3-61) and addressability to it is inserted into the Work Control Block. A DMCQ is reused from job to job.

4.1.1.1.2.4. Machine Interface Response Queue

The Machine Interface Response Queue (MIRQ) is the response queue necessary to perform the MI instruction Request I/O (see section 4.1.3.9.2 “I/O Data Structures” on page 4-61). It communicates status of the I/O operation to the Function Manager (FM) using the information coming from the MI. MI inserts a feedback record on the queue for each I/O request sent.

The FM provides a pointer to the MIRQ in the Request I/O template. Upon completion of the I/O, the FM dequeues the message into the associated space of the LUD. The FM determines from this message whether the I/O request completed without error. If an error occurred, the FM takes appropriate action according to error details recorded in the message.

MIRQ (an MI queue object) is created in system domain as part of the temporary job structure and addressability to it is inserted into the Work Control Block. An MIRQ is reused from job to job.

4.1.1.1.2.5. File Open

A file open may be explicitly originated from a user program, which, after translation, calls the API program QDMCOPEN. The main purpose of the file open is to establish the linkages necessary for file I/O. The following steps are completed by QDMCOPEN to open a file and activate the data path for I/O processing.

1. Verify the user’s UFCB. The UFCB is verified to be a UFCB and not another structure. The pointers in the UFCB are verified to not address system domain objects, system domain spaces, or write-protected spaces.

2. Determine the actual name of the file to be opened. Check for overrides for the file name specified in the UFCB.

3. Resolve to the file to be opened.

4. Allocate the resources required for the file. If opening a device file, allocate the necessary device for the file. If opening a database member, allocate the necessary data spaces.
5. Determine the location and size of the user buffers. If the user has provided buffers, keep track of the size and location of the buffers.
6. Create a copy of the ODP into the user’s space. Create a temporary space or temporary cursor object, which is a working copy of the file or member.
7. Set the file source, library, and member into the ODP. Keep track of which attributes come from overrides and which attributes come from the UFCB (i.e., record the origin of these three attributes).
8. Check authorities. By locking the resources required (i.e., the file and device or member), ensure the user has authority to these objects.
9. Add an Open Data entry to the DMCQ (ODP is now active).
10. Merge the parameters from the UFCB and any override into the active ODP. New values simply overlay the initial values copied from the prototype.
11. Link the UFCB to the ODP. Set space pointers that are located in the UFCB to point to sections of the ODP to allow the user to access information in the ODP.
12. Mark the path ready to use. Set the status field in the ODP to indicate that the file is fully open and ready for I/O.

4.1.1.1.3. Library Management

This section describes library management. The first part explains the function of the Librarian component. The second part explains the function of library objects.

4.1.1.1.3.1. Librarian Component

The librarian component has both internal and external interfaces to manage the library (*LIB) object and the objects that library object manages. The external interfaces consist of library object manipulation commands and object manipulation commands.

The librarian component has a list of all object types (MI/internal/external, type/subtype, symbolic type, descriptions and class) that reside in the system. This list is used throughout the system to perform the following functions:

- Create a library
- Delete a library and objects in the library
- Delete an object in the specified library
- Rename an object within the specified library or move an object to another specified library
- Insert an object into a specified library
- Create a duplicate object
- Display contents of libraries
- Retrieve and change object information stored in the object information repository (OIR)
- Display and change the user, system and product portions of the library list

4.1.1.1.3.2. Library Object

A library (*LIB) is made up of a context MI object (see section 3.2.1.2.2 “Contexts” on page 3-20). It is used to group related objects and to find objects by name (i.e., a directory to a group of objects). To access an object, the name, type, and library in which it exists is usually specified. The system then searches the specified library for the object name and type. The user must be authorized to the library and to the object. QSYS is the main library that contains other libraries (the only library that contains other libraries), all user profiles, and all device-related objects.

In attempting to find an object that is specified without a library name, the operating system searches according to a library search list. For each starting job, the operating system creates a default library search list that includes the system library list (contained in a system value), product library, and the user’s current library. The three types of libraries, system (shipped with the system), user, and product, are discussed below.

Each user has a library search list consisting of the system search list, product library, current library, and user-created libraries. When a library list is used to find an object, each library in the list is searched in the order of its occurrence in the list until an object of the specified name and type is found.

System Libraries organize information needed by the system to perform tasks. System libraries included in the library search list are searched before the other libraries in the list. After system installation, the system part of the library search list consists of QSYS, QUSRSYS, and QHLPSYS. QSYS contains objects that are provided as part of the operating system; QUSRSYS contains user data; QHLPSYS contains help information. The system administrator completes the setup of the system part (the user cannot change the system part) of the library search list.

Additionally, QUSRTOOL is a system library shipped with the system that contains system management utilities and tutorials. QRECOVERY is a system library that contains main storage objects gathered during recovery procedures.

The product library part of the library search list is changed by the system as the user selects commands and menus that access the libraries where the selected program is stored. While a job is running, the product libraries vary based on the function being performed. Product libraries contain programs associated with licensed products and related objects.

The current library is the default user library used for the creation of objects. Usually, it is specified in the user profile for objects created by that user. However, a user can specify the current library for a job using CL commands, in a CL program that calls the job, or from the sign-on screen. All external objects are inserted into a library. Objects created, but not qualified with a library name, are inserted into the current library. The default current library is QGPL.

User-created libraries, or user libraries, organize work on the system for specific applications. After system installation, the library search list includes the user libraries QGPL and QTEMP. QGPL is a general purpose library; QTEMP is created by the system to contain objects for each job, then the objects are deleted at the end of the job.

4.1.1.2. Internal Objects

OS/400 uses the term “internal objects” to refer to what other systems call internal control blocks. Internal objects are passive entities that contain or receive information. Internal objects are instantiations of MI objects. By convention, internal objects are created with an object subtype range of ’A0’x through ’FF’x. Internal objects may be permanent or temporary. Permanent internal objects exist until they are explicitly destroyed by the user. Temporary internal objects exist until they are implicitly destroyed by the machine when machine processing terminates or until they are explicitly destroyed by the user.

Having a specific range of subtypes for internal objects keeps internal objects from being visible to applications and other non-operating-system programs when they try to access internal objects through specific reference or generic object commands.

An example of an internal object is a system value object (SVAL). It is a subtype that identifies it as an internal system value object. Section 4.1.1.1.3.1 “Librarian Component” on page 4-11 describes how internal interfaces check to determine whether the object subtype is within the subtype range.

All internal objects are registered with the XPF librarian component (LI), which assigns them a subtype. A master list of internal objects is maintained in the PL/MI include WWLIOBJT.

Most internal objects are unavailable to untrusted users and visible only to trusted users through the restricted CL commands. These CL commands are restricted by DAC using *EXCLUDE. Internal objects are also protected from all users by being in the system domain or, if they are in the user domain, by read-only hardware-storage protection. An example of an internal object is the Authorized User Table (AUT), which contains the names of all user profiles and the one-way encrypted passwords associated with each profile name. This table is in system domain.

Certain internal objects in the OS/400 are externalized to the end user or system administrator. These fall into two categories. The first are those that are in system domain and are made visible via restricted commands to the system administrator, who can modify the contents of the object through these commands. These include the following:

- System Values: System values are used by an administrator to customize the system for the installation (see section 6.5.4.1 “System Values for Auditing” on page 6-22).
- Network Attributes: Changing the network attributes requires permission to the CL command Change Network Attributes (CHGNETA), which is restricted.
- System Reply List: The system reply list can be used to provide automated responses to operator messages. Changing the system reply list requires access to the CL commands Change Reply List Entry (CHGRPYLE), Add Reply List Entry (ADDRPYLE), or Remove Reply List Entry (RMVRPYLE).
- Network Distribution Directory: Changing the network distribution directory requires permission to the CL command Change Network Attributes (CHGNETA).
- Access path recovery time: The access path recovery time object is used to store the amount of time a user spends during an IPL on access path recovery. Changing the access path recovery time requires access to the CL commands Change Recovery for Access Paths (CHGRCYAP) or Edit Recovery for Access Paths (EDTRCYAP), which are restricted, and the *JOBCTL special privilege.

The second category of internal objects that may be visible to users, both trusted and untrusted, are in User Domain and include the following:

- An entry point table, which is a vector of authorized system pointers to many of the User Domain/System State (UDSS) and System Domain/System State (SDSS) program objects in the TCB. It is built as part of system installation by calling the MI instruction Resolve System Pointer (RSLVSP) (see section 3.2.1.2.4 “MI Pointers” on page 3-21) for each program to be included in the EPT. The EPT contains approximately 5860, entries of which approximately 660 point to User Domain program objects.

  The EPT is stored in a user domain MI space object with a hardware storage protection of read-only. It is contained in the QSYS library and owned by QSYS with public authority of *CHANGE.

  Each system pointer in the EPT contains the *CHANGE authority, which is enough authority to call the program. (The *CHANGE does not allow the user to do any more than call the program.) The EPT exists entirely for performance reasons (i.e., the MI instruction RSLVSP is not required at execution time). The EPT is addressable to each job via the WCB, which is initialized as part of job initialization.

The EPT must be rebuilt when an IPL wrap occurs or when a system program changes. Tools are available, during installation only, to the administrator to rebuild the EPT (no API accesses the EPT).

- All other user domain internal objects are process local and are used as feedback areas for status. These include WCB, ODP, and CA workspaces, which are all read-only hardware-storage protected.

OS/400 APIs are provided to access the entry point table and the WCB.

4.1.2. OS/400 Interfaces

OS/400 provides two interfaces: an interactive interface and a programmatic interface. The interactive interface is a set of CL commands that provide a means of interactively requesting services from OS/400. The programmatic interfaces provide a means for programs to request services from OS/400.

4.1.2.1. Command Language

Command language (CL) is a user interface, and a single command language statement is called a command. Commands can be entered in the following ways:

- Individually from a work station (command line)
- As part of a batch job
- As source statements to create a CL program

When a CL command is entered individually (from the Command Entry display), each command is separately processed.

CL programs are made up of one or more CL commands. The commands are compiled into a program (*PGM). (See section 4.1.3.10 “CL Compilers” on page 4-63 for more information.) Certain CL commands that are valid only in a CL program:

- Logic control functions
- Functions that allow the program to send messages to the display station
- Functions that receive messages sent by other programs

AS/400 CL commands are able to call system domain programs (e.g., CPP). Users may also create *CMD objects via the CL command Create Command (CRTCMD) from source command definition statements but these commands may call only user domain programs. Each command consists of a Command Definition Object, a command processing program (CPP), and optionally a validity checking program (VCP) or a prompt override program (POP).

The CPP is the program called when the command is entered. The CPP can be a CL or High Level Language program, or a REXX procedure. Because the system performs syntax checking when the command is entered, the CPP does not always have to check the syntax of the parameters passed to it. The CPP and VCP must accept the parameters as defined by the command definition statements.

Minimal authority needed to execute a command is *USE authority to the *CMD object, and *EXECUTE data authority to the *PGM and optional processing programs that have been specified as part of the command. The proper authority to any objects that the CPP accesses is required. These authorities are specified in appendix D of the Security Reference Manual [19].

Commands created by IBM contain an offset within the System Entry Point Table to the CPP, and optionally to the VCP and POP. These entries are system pointers that contain all the necessary authorities to use those programs.

4.1.2.2. Programmatic Interfaces

As discussed earlier in this report, at the OS/400 level, programs (*PGM objects) run in processes. Programs can “call” other programs using one of the following MI instructions:

- CALLX
- CALLPGM
- XCTL

When a program is called with one of these MI instructions, the MI instruction performs a DAC check to ensure that the user profile under which the process is running has appropriate authority to the called program. The MI instruction also performs a domain check. A process in System State can call either System Domain or User Domain *PGM objects. A process in User State can call only User Domain *PGM objects.

Procedures in bound programs can “call” procedures in the same or another bound program using one of the following MI instructions:

- CALLBP
- CALLPP

DAC and domain checks are done for procedures at bind or activation time.

As described in section 3.2.3.5.2 “Program Adoption” on page 3-70, when a program is called that has its *adopt owner authority bit* set, the calling process gains the authority of the program’s owner, including any special authorities associated with the owner. The *PGM objects in the evaluated configuration as shipped are owned by IBM-supplied user profiles (see Appendix I, “IBM-Supplied User Profiles”, on page I-1), and some of them have their adopt owner authority bit set. Whenever one of these *PGM objects is called, the calling process gains the authority of the IBM-supplied user profile along with any special authorities that it may have.

Additionally, *PGM objects have a *process state attribute* field that determines how the process state is set when the associated program is invoked. The process state attribute field can have one of three values:

- **User**: Causes the process state to be set to User State. In this section, the term *User State Program* is used to refer to any program with a process state attribute field set to User. The *PGM object creation tools shipped as part of the evaluated configuration always set the process state attribute field to User. Consequently, all user-created programs are User State programs.
- **System**: Causes the process state to be set to System State. In this section, the term *System State Program* refers to any *PGM object with a process state attribute field set to System.
- **Inherit**: Causes the process state to be unchanged. If the process is in System State when the program is called, it remains in System State. If the process is in User State when the program is called, it remains in User State.

Because *PGM objects can be either User Domain or System Domain, there are, therefore, six possible combinations (listed below):

- User Domain System State (UDSS)
- System Domain System State (SDSS)
- User Domain User State (UDUS)
- System Domain User State (SDUS)
- User Domain Inherit State (UDIS)
- System Domain Inherit State (SDIS)

Of these six possible categories, the evaluated configuration contains programs in four categories. Each of these four categories is described in the sections that follow.

4.1.2.2.1. User Domain Inherit State Programs

OS/400 contains approximately 537 User Domain Inherit State (UDIS) programs. None of these programs have their program adoption flag set and, therefore, cause a calling process to gain the authority of an IBM-supplied user profile.

Since UDIS programs are in User Domain, they can be called from both System State and User State. Since they are Inherit State, they do not cause a state transition when called from either State.

4.1.2.2.2. User Domain User State Programs

OS/400 contains approximately 209 User Domain User State (UDUS) programs. Twenty one of these programs have the program adoption flag set and, therefore, cause a calling process to gain the authority of an IBM-supplied user profile along with its associated special authorities.

4.1.2.2.3. System Domain System State Programs

OS/400 contains approximately 4172 System Domain System State (SDSS) programs. Since they are in System Domain, these programs cannot be called from User State. Since they can be called only from other System State programs (and since only IBM-supplied programs are System State), SDSS programs “trust” that parameter validation has already been done and, therefore, perform no parameter validation on their input parameters.

4.1.2.2.4. User Domain System State Programs

OS/400 contains approximately 690 User Domain System State (UDSS) programs, which IBM refers to as *Application Programming Interface (API)*. Approximately 220 of these programs have the program adoption flag set, and therefore, cause a calling process to gain the authority of an IBM-supplied user profile along with its associated special authorities.

Since the APIs are in User Domain, they can be called from User State using the unblocked MI instructions. Since they are System State programs, the APIs can call the SDSS programs.

Since the UDSS programs can be called from user-developed code (i.e., “untrusted” code), they must perform *parameter validation* on any parameters that are passed to them as described in section 4.1.2.2.4.1 “Parameter Validation” on page 4-16.

4.1.2.2.4.1. Parameter Validation

AS/400 relies on both object domain and hardware-storage protection (in addition to DAC) to protect itself and the objects under its control. Whenever a process is running in System State, the object domain and hardware-storage protection mechanisms are ignored. That is, the process can access System Domain objects, and it can write to write-protected objects.

When the APIs are called by user-developed code, they cannot “trust” that they will not be passed pointers to System Domain objects or to write-protected User Domain objects. Consequently, they must protect themselves (and the SDSS and UDIS programs they subsequently call) against being passed pointers to System Domain objects or write-protected User Domain objects by performing a series of checks that are referred to as *parameter validation*. These checks are described below:

- To prevent parameters from being modified by non-OS/400 code (i.e., User State code) after they have been validated, the UDSS program first copies all parameters from UD storage to SD storage.
- The UDSS program checks that each system pointer points to UD storage and that each space pointer points to non-write-protected UD storage. The failure of either of these checks is an auditable event.
- Data, instruction, invocation, procedure, label, and suspend pointers can be passed outside a bound program. However, if one of these is a parameter when a system or space pointer is expected, it causes an error.
- The UDSS program validates all pointers. That is, all pointers within a structure or pointed to by other pointers are checked to verify that the objects being pointed to are of the expected object type.
- The UDSS program checks each scalar parameter for the proper range, value, and syntax.

4.1.2.2.5. Kernel Mode

An above-MI kernel environment was defined for AS/400 in V3R1. The kernel environment is a fast way to invoke certain above-the-MI functions. The kernel functions that affect the TCB are primarily to support the Integrated File System (see section 4.1.3.2 “Integrated File System” on page 4-19).

The kernel environment is a set of procedures that are activated at IPL time in what is called the Shared Activation Group (SAG). The service programs or procedures in the SAG are activated from a list provided by OS/400. The list is then used to create a table for invoking the procedures. This system-defined table, known as the Kernel Mode Routing Table (KMRT), is constructed as an encapsulated list of procedure pointers by SLIC as a part of the MI instruction MODMATR execution. OS/400 defines the procedures and order in which the procedures appear within the Kernel Mode Routing Table. The KMRT can contain 0 to 4,064 entries. Each entry is a 16-byte procedure pointer. On each system IPL, the internal storage for this table is set to the empty state; and thus, it does not contain valid information until built by the MI instruction MODMATR. The SLIC also verifies each procedure pointer to ensure it is a valid pointer.

The Kernel environment (SAG) is entered via the NMI instruction System Call (SYSCALL), an interface like the NMI instruction Call Procedure Pointer (CALLPP). The MI instruction SYSCALL provides the mechanism for accessing procedures that execute in the kernel environment. SYSCALL indicates to the calling process that it is running in kernel environment, i.e., the shared activation group. An integer specified by the first operand of the MI instruction SYSCALL is an index value into the KMRT system table. Operand 2 is a space pointer to a user-defined structure containing the argument list from the calling program. This argument list is validated if changing from user to system state, in the same way as if it were the arguments for a call from a user state to a system state program. The kernel procedure is placed on the user process stack and the automatic storage is allocated from the user process system state storage.

Kernel functions are system state procedures that execute in the shared activation group. Kernel procedures run in System State so that untrusted users cannot access the kernel program storage in the shared activation group. The MI instruction SYSCALL is blocked and is only used in OS/400-provided function. The displacement into the KMRT is coded as a constant in the system code so that the displacement cannot be modified at execution time.

The MI instructions System Call Procedure (SYSCLLPP) and System Call Program (SYSCLLPG) provide a mechanism for allowing system functions that execute in the kernel environment (SAG) to access procedures or programs not intended to execute in the kernel environment. SYSCLLPP and SYSCLLPP set the execution environment of the calling process back to the user. The setting of the process execution environment to user causes actions that were held pending for the process while the process execution was in the kernel to be taken at the next MI instruction boundary. These actions include the scheduling of event handlers that may not be scheduled while in the SAG or for events that may have occurred while the process was executing in the kernel.

4.1.3. OS/400 Functional Areas

This section describes the functional areas that comprise OS/400. These areas include the following:

- Storage Management
- Integrated File System
- Job Management
- Spool
- Command Analyzer
- Request Processors
- Message Handling
- Event Management
- I/O Management
- CL Compiler
- REXX Interpreter
- Debugger
- System Values
- Exit Points, Exit Programs, and User Exits

4.1.3.1. Storage Management

In the single-level storage model used by the AS/400, the operating system has minimal direct interfacing with storage management (SM) and the end user has even less. (See section 3.2.3.3 “Storage Management” on page 3-47 for details on use of main and auxiliary storage.) The only interfaces available are designed for performance tuning and availability. Three mechanisms influence SM operation: main storage pools, setting access states, and auxiliary storage pools.

The only interface to main storage pools is through the work management component. The system administrator divides the system main storage resources into pools by using the CL command Create Subsystem Description. These restricted commands on AS/400 allow the trusted administrator to tune the system. The concept of pools means no more than dividing the real storage into partitions to be allocated to various types of jobs, such as interactive and batch.

The administrator may define up to 16 pools, including the IBM-provided pools for *BASE, shared storage pools (*MACHINE, *INTERACT, *SPOOL, and *SHRPOOL1...10), and user-defined private pools. A *shared storage pool* is a pool in which multiple subsystems can run jobs.

Storage pools include the following:

- *MACHINE is the system storage pool for IBM-provided programs. The size of this storage pool is specified in the system value QMCHPOOL. No user jobs run in this storage pool.
- *BASE is the base storage pool that contains all unassigned main storage on the system; that is, all storage that is not used by the machine storage pool or by another pool. The base storage pool is used for batch work and system jobs.
- *INTERACT is the interactive storage pool used for interactive jobs.
- *SPOOL is the storage pool used for spooling readers and writers.
- *SHRPOOL1 through *SHRPOOL10 are storage pools available for general use.

For shared pools, the size and activity level can be changed. For the machine pool, only the size can be changed. For the base pool, only the activity level can be changed. Changes to shared pools take effect immediately if the shared pool is active and sufficient storage is available.

A *private storage pool* cannot be shared by multiple subsystems. A private pool contains a specified amount of storage to be used by only one subsystem. As many as 14 private pools can be allocated for use in active subsystems (but there is a limit of 10 per subsystem).

A system job, called the *subsystem monitor job*, controls all the activity in a subsystem. The subsystem monitor job is started when the subsystem is started, but does not count toward the activity level of the subsystem. The shipped default causes the storage for the subsystem monitor to come from the base storage pool.

The *activity level* of a storage pool is the number of jobs that can run at the same time in a storage pool. Once the maximum activity level for a storage pool has been reached, additional jobs needing the storage pool are automatically put on the SRQ, where they wait for a message making them eligible.

An option for tuning, used by OS/400 for multi-segmented objects, is the ability to force objects into specific main storage pools or to purge them from main storage, as required. This is accomplished by the MI instruction Set Access State (SETACST). The operating system can also control the paging of groups of objects by using the Access Group MI commands. This defines a group of objects to be paged in and out together. A set of process objects usually form an Access Group.

Besides the main storage pools discussed above, the auxiliary storage comes in one large pool called ASP1 (auxiliary storage pool one). If a user installation requires certain libraries or other objects to be on a specific DASD device, users can define ASPs (numbers ASP2-16). ASPs can be created only by a trusted user using the dedicated service tool and only the trusted administrator can allocate certain DASD devices to user-defined ASPs.

4.1.3.2. Integrated File System

The integrated file system function provides a hierarchical directory and stream file function on AS/400. The hierarchical directory and underlying file system support the semantics of DOS, OS/2, and POSIX as well as provide a directory oriented interface to AS/400 existing library structure. This function is designed to allow for porting applications to the AS/400 from other systems. This function was added to the TCB in Version 3 release 2. The integrated file system provides an umbrella of support for a number of file systems that manage files, directories, libraries, and other objects primarily through interfaces like POSIX.

The integrated file system supports the following file systems in the AS/400 TCB:

- A root ’/’ file system, which provides full support for the stream file and hierarchical directory structure. The root file system has the characteristics of the Disk Operating System (DOS) and the OS/2 file systems in that object names are not case sensitive.
- An open systems file system, QOpenSys, which, like the root file system, provides full support for the stream file and hierarchical directory structure. This file system is designed to be compatible with the POSIX-based open file standards. In addition, it supports case-sensitive object names.
- A library file system, QSYS.LIB. It supports the AS/400 library structure. The QSYS.LIB file system provides access to database files and all other objects that are managed by the AS/400 library support. Through an architected interface called the VFS/Vnode interface, it maps to existing (pre-V3R2) AS/400 libraries and the objects within them. That is, it provides a way to access pre-V3R2 objects through a directory.

Specific functions are available in the integrated file system to support the root and open file (QOpenSys) systems. These include the following:

- A Directory Object Manager for directories
- A Byte Stream File Object Manager that is common to the integrated file system
- A set of generic object management commands for managing the objects and directories in the root and QOpenSys file systems
- A set of APIs based on POSIX and UNIX standards to create, delete, open, close, read, write, and manage objects in directories
- The save/restore and reclaim system functions for stream file and directory objects, which are supported through the common system service

Architecturally, the integrated file system consists of the following three logical entities, which are each managed within the indicated configuration items and are illustrated in Figure 4-2. “Integrated File System Structure” on page 4-21.

- A Hierarchical Directory (component P0D)
- The Logical File System (LFS) (component P0L)
- The Physical File System (PFS) (component P0F)

The Integrated File System implements an architecture known as the Virtual File System/Virtual Node (VFS/Vnode) Interface. This architecture provides a file type independent set of functions that may be performed against the objects in the physical file systems.

Four named, external object types are specifically and uniquely supported by the integrated file system. For other objects on the AS/400, support is provided by the LFS as a front-end interface to their normal AS/400 interfaces with limited directory support and only symbolic links in, not out. The integrated file system specifically supported external XPF objects are as follows:

- *STMF - Stream File
- *SYMLNK - Symbolic Link
- *DIR - Directory
- *BLKSF - Block Special File

The Stream File (*STMF), Symbolic Link (*SYMLNK), and Block Special File (*BLKSF) are subtypes of a single MI object called the Byte Stream File MI Object, which is owned by SLIC component P0F.

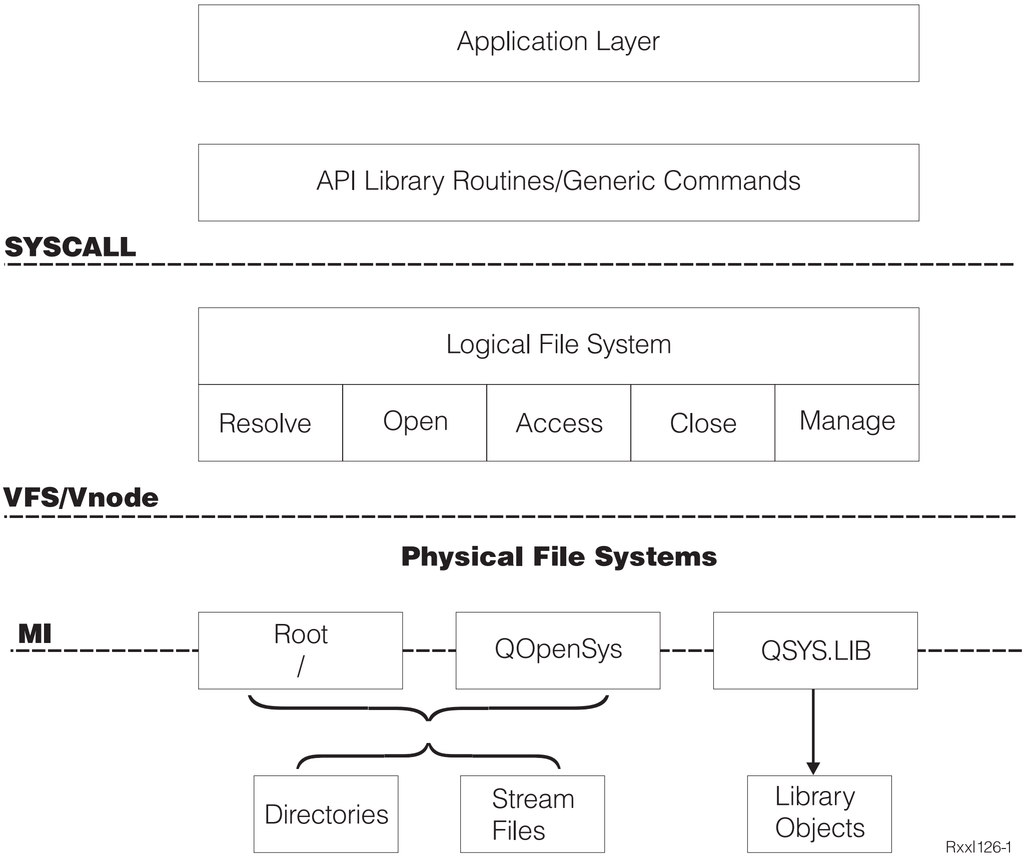


Figure 4-2. Integrated File System Structure

The directory component (P0D) owns the Directory (*DIR) object. The *DIR object is based on the Data Space Index MI object and managed via existing MI instructions that operate on the Data Space Index.

Unlike other XPF objects, these four external objects can be contained only in directories. They can not be contained in libraries. They are externalized through generic commands and integrated file system APIs.

Conceptually, the directory used by the integrated file system is similar to context (library) management, where a machine index subsumes the context’s (library) function.

During the IPL when the system is being installed, the directory component establishes the root, notroot, and the dead directories. Then a system routine validates that the base root file system exists and is usable.

The Logical File System (LFS) provides the POSIX APIs and additional AS/400 unique APIs. The LFS provides services much like those provided by Common Data Management in terms of the overall management of open files for a process. The file system interfaces are based on industry standards from POSIX, but contain additions such as accepting “name structures” instead of strings for names. External APIs use strings while internal interfaces use structures. The “name structures” allow the specification of the coded character set identifier, the language, and the country of the name, as well as the name itself.

The compiler execution time library routines invoke the LFS functions by using the MI instruction System Call (SYSCALL) (see section 4.1.2.2.5 “Kernel Mode” on page 4-17). The functions invoked by SYSCALL perform file descriptor (file handle) functions, maintain open instances, and invoke the VFS/Vnode Interface to pass requests to one of several physical file systems (PFS).

The physical file systems implement the VFS/Vnode internal Interface. This interface defines two principal structures:

- The VFS Structure defines an instance of a PFS.
- The Vnode Structure defines an object managed by a PFS.

The vnode is a structure that represents an object stored in the file system. Vnodes allow access to those objects using a consistent interface that is not concerned with the underlying physical representation, implementation, or location of the object. As objects in the file system are referred to, vnodes are created and put into a linked list of cached vnodes, allowing fast access to recently accessed objects in the file system. The cache of vnodes is managed such that when an object is no longer accessed periodically, its vnode becomes a candidate for reuse to represent another object in the file system. The origination, management, and deletion of vnodes is transparent to the user of integrated file system functions.

4.1.3.2.1. MI Interfaces and SLIC Support

As introduced above, a new MI stream file object was defined to support the integrated file system. Three MI instructions are provided to manage the stream file object. Additionally, an asynchronous I/O task performs the actual scheduling of the object's data to DASD. The three blocked MI instructions are as follows:

1. File Server/System Router (FSROUTE)
2. File System Read (FSREAD)
3. File System Write (FSWRITE)

The MI instruction FSROUTE is used to invoke the File System to perform a variety of functions, including create, destroy, lock management, authorization, and copy.

The MI instruction FSREAD provides for the reading of data contained in a stream file object. The parameters passed to the FSREAD instruction consist of the architected parameters for the vn_read vnode operation, except that the parameter that points to the credentials for the calling process is not included.

The MI instruction FSWRITE is invoked by the File System vn_write support. The parameters passed to this support consists of the parameters for the vn_write vnode operation, except that the parameter which points to the credentials for the calling process is not included.

In addition, other generic MI instructions recognize and support the Byte Stream File MI object, such as Materialize Pointer (MATPTR) and Materialize System Object (MATSOBJ).

4.1.3.2.2. The Root and Open File Systems and the Security Model

To be consistent with applications that were ported from POSIX-based systems, authority definition mapping is performed between POSIX permissions and AS/400 authorities.

POSIX defines a file mode associated with an object, which is used to declare the data authority that users of the object have. The file mode contains nine permission bits. These bits are split into three sections:

OWNER	The authority that the owner of the object has
PRIMARY GROUP	The authority the primary group of the object has
OTHER	The authority that 'public' has to the object

Each section has three bits, which declare the data authority that a user belonging to that section has to the object. The three bits may appear in any combination with each other. They are defined as follows:

READ	The user can read from the object.
WRITE	The user can write to the object.
EXECUTE	The user can execute or search the object.

These Read, Write, and Execute bits map to AS/400 authorities as follows:

READ (*R)	Operational + Read
WRITE (*W)	Operational + Add + Update + Delete
EXECUTE (*X)	Operational + Execute

When creating a file or directory using the APIs, the authorities for the object come from two places:

1. The file system APIs require that the application declare the authority in the file mode, which is passed as a parameter to the API. The AS/400 data authorities are set according to the bits specified in the mode, as defined above.
2. The AS/400 object authorities (object management, existence, alter, and reference) are set according to the particular physical file system managing the object; that is, if it is an AS/400 library object, they are set as in the past. If it is a directory, symbolic link, or stream file, they are inherited from the directory in which the object is created.

If an object authority needs be set or reset, the owner or other authorized user must use the CL command Change Authority (CHGAUT) or the chmod() API to set or reset the appropriate authority bits.

To be consistent with POSIX, integrated file system APIs run against QOpenSys and root file system objects run only with the authority of the profile under which the job is running. That is, integrated file system APIs do not adopt the owner’s profile.

4.1.3.3. Job Management

Work management is the OS/400 component responsible for managing jobs and system values. This section discusses the various types of OS/400 jobs and their associated data structures.

There is a one-to-one relationship between an active job and a process. However, a job has its own unique identifier and data structures that are managed exclusively at the OS/400 level.

All jobs on the AS/400 run under a user profile and, optionally, up to 16 group profiles. Jobs are associated with a user via a system pointer to a user profile object from the job data structure. This is discussed further in this section. The user profile is associated with the job as part of process initiation. The user profile that the job starts under can either be specified at job start up time or comes from a job description object or an entry in the subsystem description.

Jobs are uniquely identified by a sequential job ID number. The job ID consists of the device name, user profile name, and a 6-digit number.[1] Each time a job is started, the 6-digit number is incremented. Job ids are maintained in a system table.

[1] The system maintains an index of all jobs on the system. This index prevents the job ID from being the same as one for an existing job.

If the user profile does not exist on the system, then the job does not start.

The following list identifies the different types of jobs that can be started on the system. All of these jobs are considered user jobs except for jobs in the system job category. The job type is an attribute stored in the job structure.

- Interactive Jobs: Interactive jobs are started as a result of successful login. The interactive job is started under the login user profile. Details of login are provided in section 6.1 “Identification and Authentication” on page 6-1.
- Batch Jobs: The user attempting to start the batch job must have *USE authority to the user profile under which the batch job will run or the job will not start. The subsystem monitor resolves to the user profile name to get addressability to the user profile. Batch jobs are started as a result of the CL command Submit Job.
- Autostart Jobs: When a subsystem is started by an authorized user, any autostart jobs that are configured in the subsystem description object start. The user profile under which the job runs is defined by the job description object that is named in the autostart job entry of the subsystem description object. The subsystem monitor resolves to the user profile name to get addressability to the user profile.
- Writer Jobs: When a printer writer is started by an authorized user, the writer job starts and runs under the user profile QSPLJOB. The subsystem monitor issues a resolve instruction to get addressability to the QSPLJOB user profile. A detailed description of writer jobs is discussed in the section 4.1.3.4 “Spool” on page 4-35.
- Reader Jobs: When a reader job is started by an authorized user, the reader job starts and runs under the user profile QSPLJOB. The subsystem monitor issues a resolve instruction to get addressability to the QSPLJOB user profile. A detailed description of reader jobs is provided in the section 4.1.3.4 “Spool” on page 4-35.
- System Jobs: Several jobs are started at IPL by the operating system code. These jobs all run under the QSYS user profile. These jobs (i.e., QSYSARB, SCPF, Subsystem Monitors, see Appendix H, “System Jobs”, on page H-1) perform various system tasks while the system is up and running.

All job types other than system jobs run within subsystems. If the user does not have adequate authority to the Job Queue (JOBQ), printer device, and user profile, then the job does not start.

4.1.3.3.1. Job Structure

Jobs have three phases on the AS/400 system:

- Phase 1 - Job Queue
- Phase 2 - Active
- Phase 3 - Output Queue

In Phase 1, jobs are placed in a job queue and are ready to run. Jobs can be placed on a job queue by the CL command Submit Job or the CL command Start Database Reader. Placing jobs on the JOBQ is discussed further in section 4.1.3.4 “Spool” on page 4-35. In Phase 2, jobs are in the Active Phase and have an associated process. In Phase 3, job processing has ended, but there is still spooled output associated with the job and that output is handled by a system job. The relationship of the job structure to job phases is illustrated in Figure 4-3. “Job Phases” on page 4-25. The job waits on a queue until the queue is allocated to an active subsystem.

Each job has associated internal objects that comprise the permanent and temporary job structure. The permanent job structure exists for the entire duration of the job (Phase 1 - end of Phase 3). A temporary job

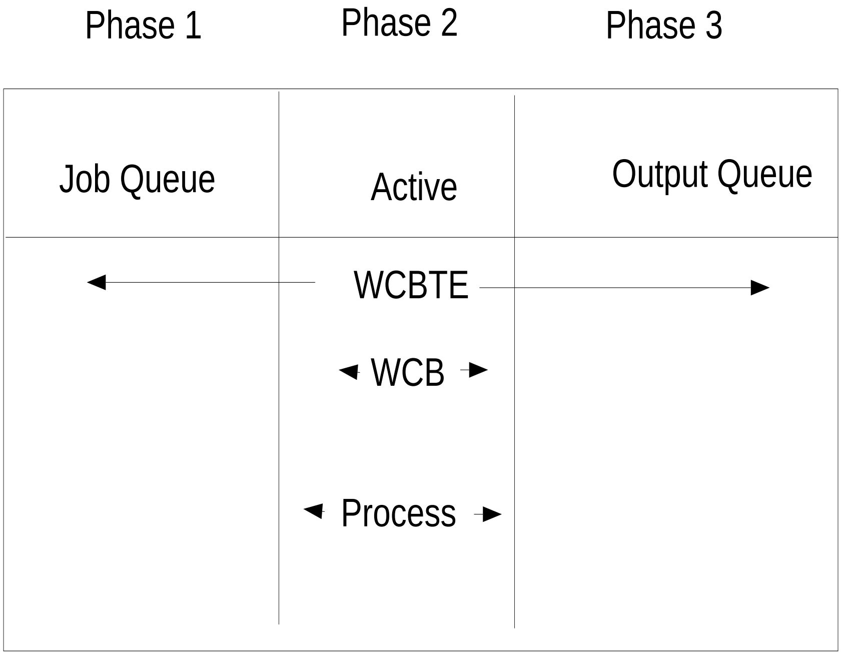


Figure 4-3. Job Phases

structure exists only while the job is active (e.g., in Phase 2). The permanent job structure, like other permanent objects, survives IPLs.

The permanent job structure consists of the Work Control Block Table (WCBT) Entry. Each job on the system has an entry in the WCBT. There is one WCBT on the system, which consists of a table of entries that provides part of the permanent job structure for each job that is active, on a job queue, or has spooled output. The Work Control Block Table Entry (WCBTE) is assigned to the job when it is started (on job queue for batch jobs, at sign-on for interactive jobs). The WCBT entry contains the following:

- Job Message Queue (JMQ) pointer - system domain; contains a running log of job activities including commands that are issued.
- Spool Control Block (SCB) pointer - system domain; contains information about files used by Spooling component.
- Job location pointer - system domain; points to the current location of a job. In phase 1, this is the JOBQ.
- Local Data Area (LDA) pointer - user domain; this is the user data area for use within a job.
- Other fields including the job name, user profile name, job type, and indications of the job status (e.g., transfer in progress).

The WCBT entry is assigned to a job by the Subsystem Monitor job. The WCBT entry is in system domain, as are the JMQ, SCB, and JOBQ. The LDA is in user domain.

The permanent and temporary job structures are illustrated in Figure 4-4. “Job Structure Figure” on page 4-26.

The temporary job structure consists of the WCB. The WCB is a temporary structure that is used only while a job is active. When a job enters the active phase, a process is started for the job. The WCB is the associated

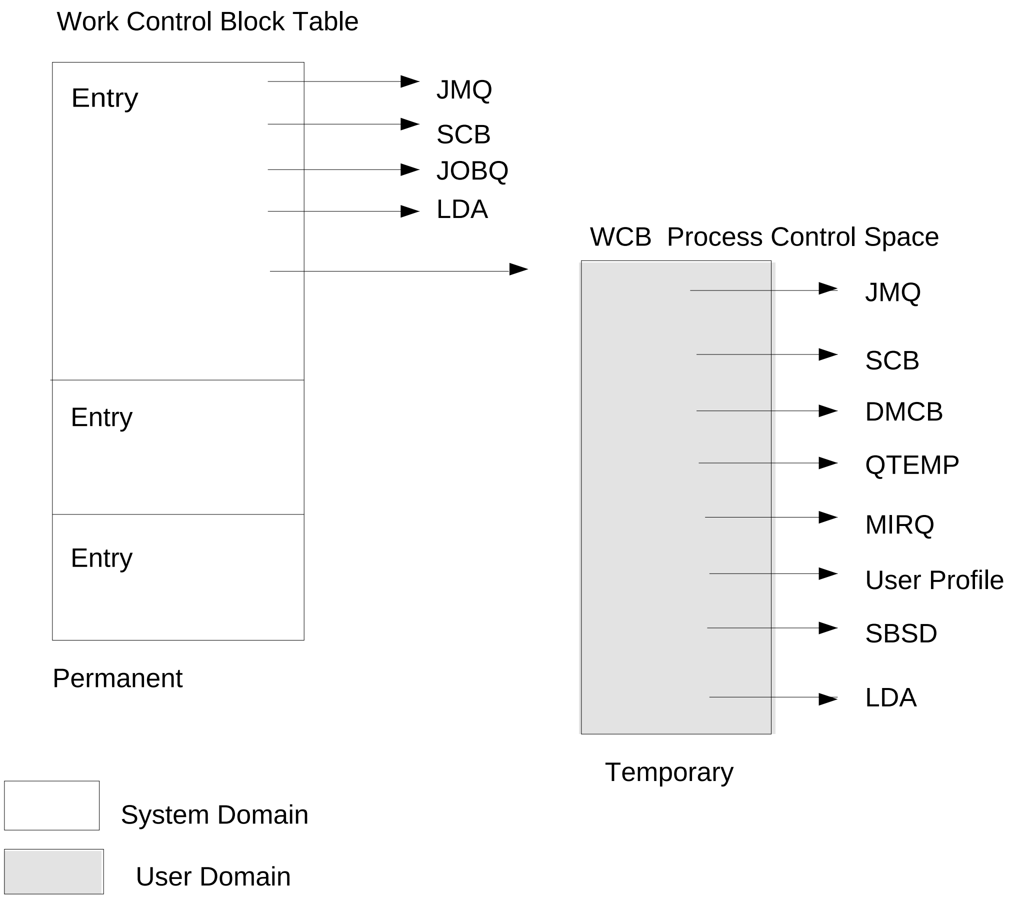


Figure 4-4. Job Structure Figure

space of the Process Control Space MI object. The WCB is initialized during job initialization. It is a control block which is allocated in user domain storage. It is created as hardware write-protected. In addition to the WCB, other internal objects are created as working areas for various components. The WCB contains pointers to various work areas (i.e., SCB, JMQ, LDA), the WCBT pointer and offset (pointer used to locate the WCBTE for job), MI Response Queue (MIRQ) User Profile pointer, Job Locator pointer (in phase 2, this points to the Subsystem Description), QTEMP pointer, and Language and Utility (LU) area pointer.

The LU work area and QTEMP are created in user domain, as is the WCB. The other parts of the temporary job structure are in system domain. A Data Management Control Queue (DMCQ) is created to support Data Management functions, as discussed in section 4.1.1.1.2 “Data Management” on page 4-9. A MIRQ is also created as part of the temporary job structure. The MIRQ is used by MI for above-MI and below-MI communications.

When a job has finished its processing, two options are possible:

1. If the job has spooled output, then the job remains in Phase 3 (output queue) until the output is removed from the queue.
2. The job ends.

The temporary job structure is kept only while the job is active. Information contained in the WCB that is necessary for Phase 3 is copied to the WCBT entry. The WCBT entry is released after all output associated with the job is removed from the queue. The Work Management component is responsible for keeping track of available WCBT entries and free WCB spaces for reallocation to other jobs.

A temporary library (QTEMP) is also part of the temporary job structure. QTEMP is a temporary library used during the life of the job. It is addressable only within the job via the pointer in the WCB. Its address cannot be resolved so other jobs cannot gain addressability to it. Each job has its own QTEMP library. When *USRSPC, *USRQ and *USRIDX external objects are created in QTEMP, they are created as user domain objects. This is enforced by the commands that create objects.

When the job is terminated, the Work Management component deletes all objects contained in QTEMP.

4.1.3.3.2. Subsystem Monitor Jobs

A subsystem monitor job manages other jobs. The system allows a user with *JOBCTL special authority to specify the number of subsystems and what types of jobs run in the subsystem.[2] The run-time characteristics of a subsystem are defined in the Subsystem Description external object (*SBSD). All subsystems have a subsystem description.[3] Subsystem jobs are a type of system job that are started by the System Arbiter #4 job. The System Arbiter #4 job starts subsystems in response to the CL command Start Subsystem. The system comes configured with a default set of subsystems and associated subsystem descriptions. These can be modified by a trusted user. The subsystem description consists of three parts:

- Subsystem attributes
- Work entries (sources of work)
- Routing entries

4.1.3.3.2.1. Subsystem Attributes

Subsystem attributes provide the overall characteristics of the subsystem. These attributes include the following:

- Maximum number of jobs that can be active in the subsystem at the same time
- Sign-on display file (to use for interactive work) - this is a DDS file with the sign-on format. IBM provides a default sign-on screen that can be modified. Modifications are limited to ensure that identification and authentication (IA) information is included as part of the display file.
- System library entry - used as the first library in the library list or NRL

4.1.3.3.2.2. Work Entries

Work entries specify how jobs can start in a subsystem. The following list specifies the work entries supported in the evaluated configuration:

- Autostart job entry: identifies the jobs to start as soon as the subsystem starts. These jobs are started when the subsystem starts.
- Workstation entry: identifies the display devices from which to take work for interactive jobs. This entry can be individual, *DEVD names, or *DEVD types.
- Job queue entry: identifies the job queues from which to take work for batch jobs

[2] This is done by modifying the subsystem description.

[3] If the subsystem description is removed, a Subsystem description not found error occurs and the subsystem does not start.

4.1.3.3.2.3. Routing Entries

Jobs can be started via routing entries. A routing entry in a subsystem description specifies the first program to be called when the job starts in the subsystem (e.g., a request processor section 4.1.3.6 “Request Processors” on page 4-51).

4.1.3.3.2.4. Subsystem Job Initiation and Termination

Subsystems are started in the following ways:

1. As part of IPL, the controlling subsystem and the work management subsystem are started via an OS/400 routine, which performs the start subsystem function.
2. Other subsystems are started via the CL command Start Subsystem. A user must have *JOBCTL special authority and authority to the subsystem description to start a subsystem.

Figure 4-5. “Starting Subsystems” on page 4-29 illustrates the steps involved in starting subsystems.

For subsystems, System Arbiter #4 is responsible for assigning the WCBTE data structure associated with jobs. After the WCBTE has been assigned, the System Arbiter #4 allocates the WCB temporary job structure. The System Arbiter #4 is also responsible for initializing the Process Definition Template (PDT). The PDT defines the attributes of a process that is passed to the MI instruction Initiate Process. As described in section 3.2.3.4 “Process Management” on page 61, the PDT contains the following system pointers:

- User Profile
- PCS and associated Work Control Block
- Name Resolution List
- Initiation Phase Program
- Problem Phase Program
- Termination Phase Program
- Process Default Exception Handler
- Queue Space
- Process Access Group

The last activity of the System Arbiter #4 in this process is to initiate the subsystem process. At that point the subsystem job begins activity.

When the subsystem starts, it runs the initiation phase and problem phase programs. The initiation phase program sets up event monitors and builds the temporary job structure that was previously allocated by System Arbiter #4. During the problem phase, the subsystem job performs the intended job activity.

When a subsystem is started, it attempts to allocate all display devices in the subsystem description. The following situations may occur during the time the subsystem starts:

- If the device is not varied on, the subsystem cannot allocate it.
- If the device is varied on and has not been allocated by any other subsystem, the subsystem can allocate it and display the Sign On display.
- If the device is varied on and has been allocated by another subsystem and is at the Sign On display, a second subsystem can allocate the device from the first subsystem and display the Sign On display.[4]

Subsystems are ended by issuing the CL command End Subsystem.

[4] A device can be allocated only to one subsystem at a time

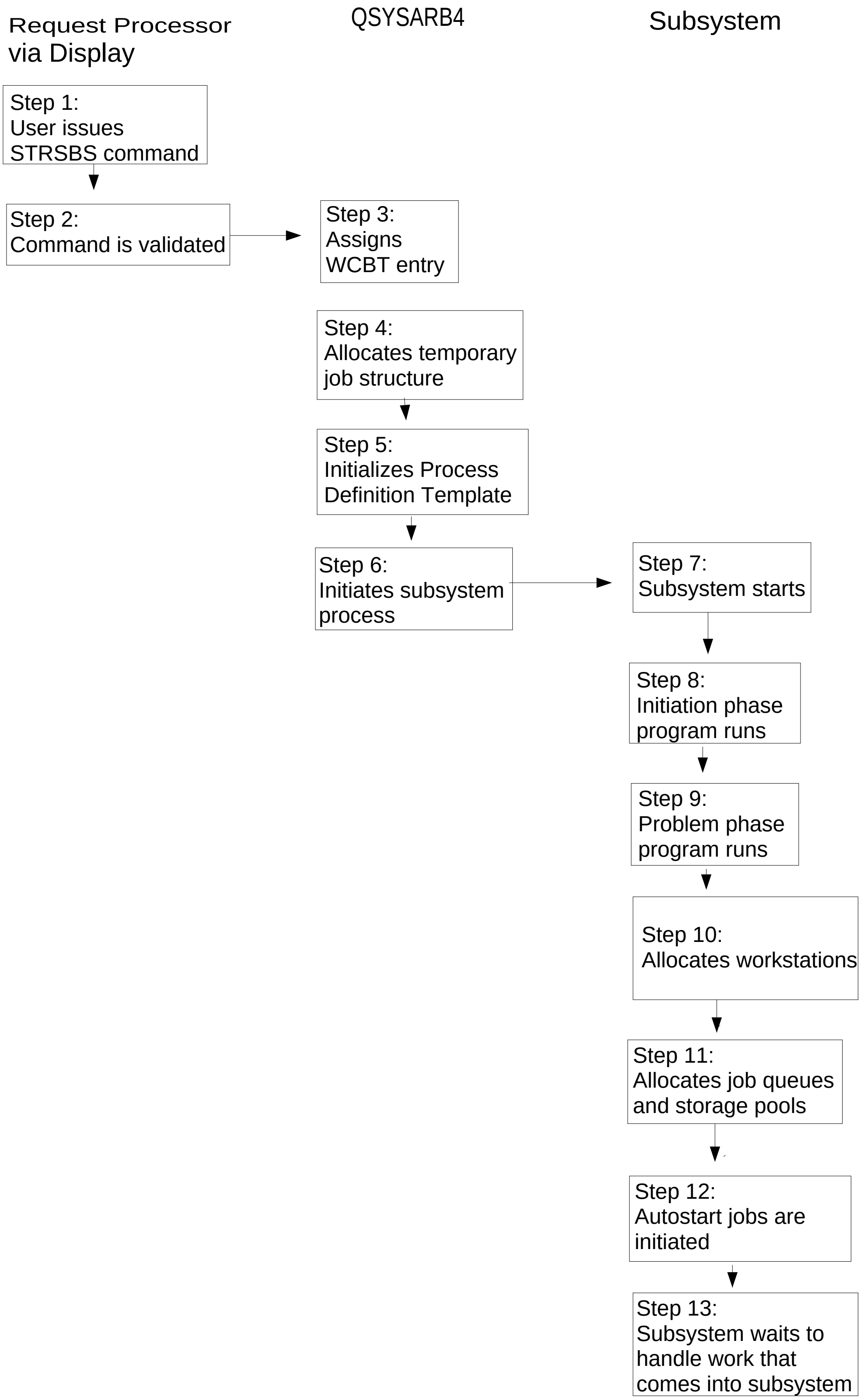

Request Processor via Display
QSYSARB4
Subsystem
Step 1: User issues STRSBS command
Step 2: Command is validated
Step 3: Assigns WCBT entry
Step 4: Allocates temporary job structure
Step 5: Initializes Process Definition Template
Step 6: Initiates subsystem process
Step 7: Subsystem starts
Step 8: Initiation phase program runs
Step 9: Problem phase program runs
Step 10: Allocates workstations
Step 11: Allocates job queues and storage pools
Step 12: Autostart jobs are initiated
Step 13: Subsystem waits to handle work that comes into subsystem

Figure 4-5. Starting Subsystems

Figure 4-6. “End Subsystem” on page 4-30 illustrates the events that occur as part of ending a subsystem.

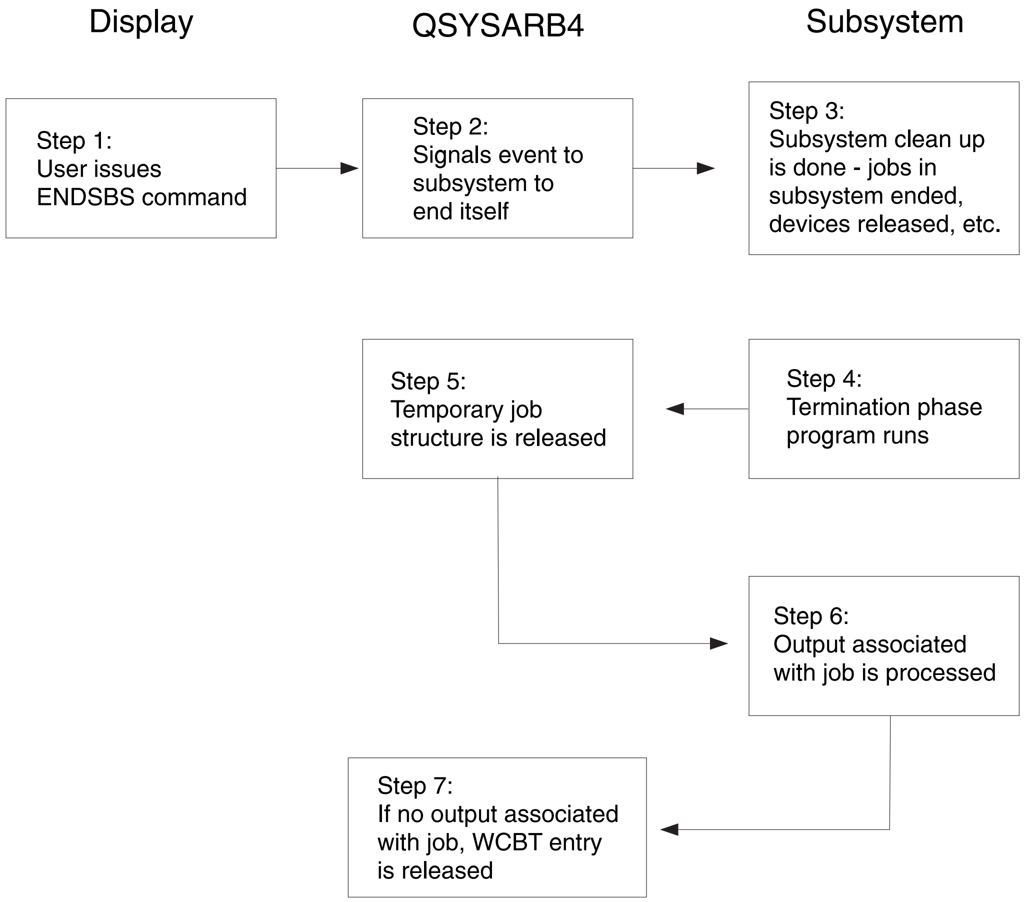

Display
QSYSARB4
Subsystem
Step 1: User issues ENDSBS command
Step 2: Signals event to subsystem to end itself
Step 3: Subsystem clean up is done - jobs in subsystem ended, devices released, etc.
Step 4: Termination phase program runs
Step 5: Temporary job structure is released
Step 6: Output associated with job is processed
Step 7: If no output associated with job, WCBT entry is released

Figure 4-6. End Subsystem

4.1.3.3.3. Interactive Jobs

An interactive job starts when a user signs on to the system from a display station and ends when the user signs off and all spooled output is removed from the queue. Figure 4-7. “Interactive Job Startup” on page 4-31 illustrates the steps involved in starting an interactive job.

When an interactive job is started by a user signing onto the system, the subsystem first looks in the workstation entry for the job description to get the attributes for the interactive job. If the workstation entry specifies *USRPRF for the job description, the job uses the information from the user profile.

After the subsystem gathers the job’s attributes, it checks the job description for the routing data. The subsystem uses the routing data to locate a routing entry in the subsystem description. The routing entry provides information about which pool the job will use, which routing program will be used, and other runtime attributes.

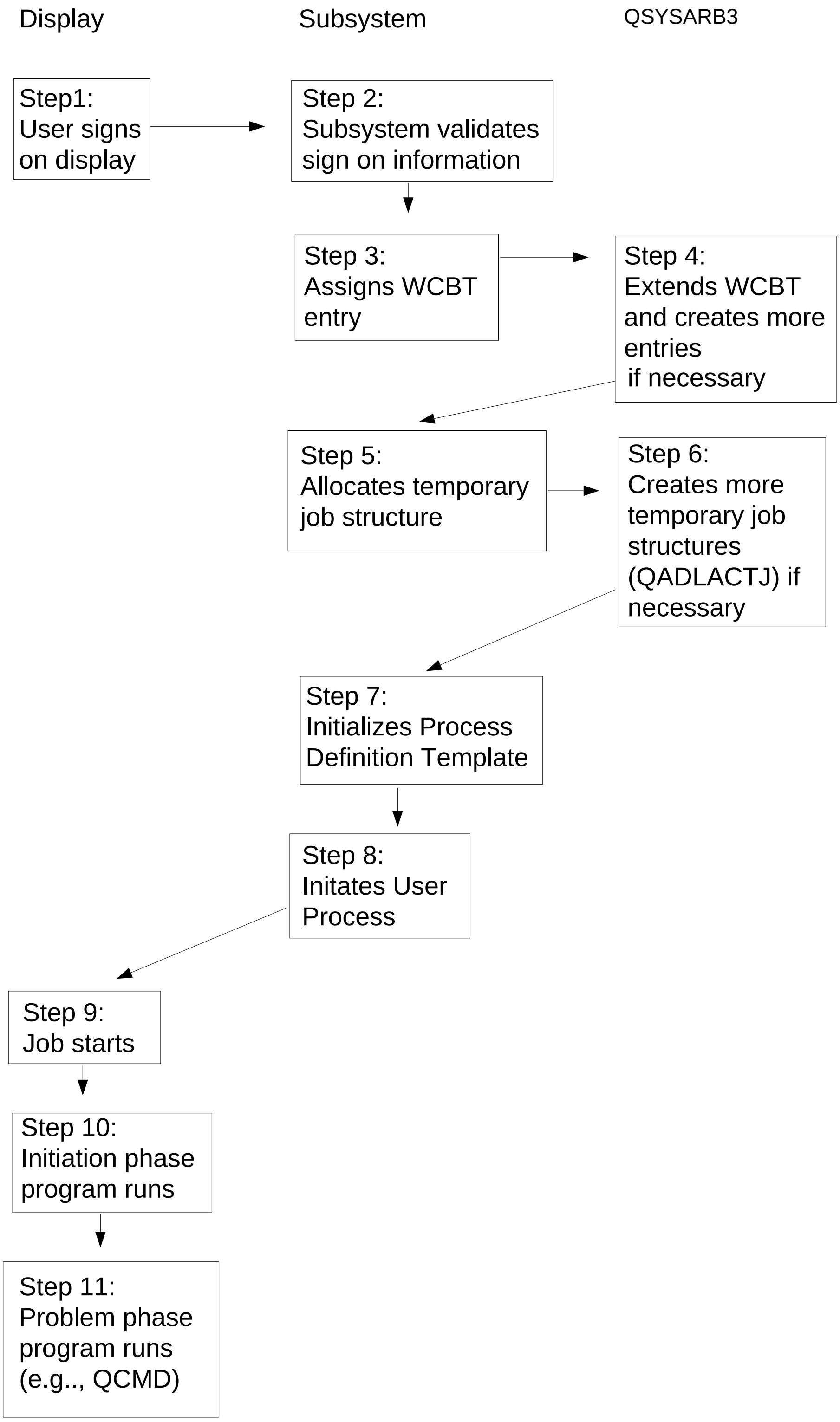

Display
Subsystem
QSYSARB3
Step1: User signs on display
Step 2: Subsystem validates sign on information
Step 3: Assigns WCBT entry
Step 4: Extends WCBT and creates more entries if necessary
Step 5: Allocates temporary job structure
Step 6: Creates more temporary job structures (QADLACTJ) if necessary
Step 7: Initializes Process Definition Template
Step 8: Initates User Process
Step 9: Job starts
Step 10: Initiation phase program runs
Step 11: Problem phase program runs (e.g.., QCMD)

Figure 4-7. Interactive Job Startup

To start an interactive job successfully, preliminary authority checking is performed before the job actually starts. Certain steps illustrated are further expanded:

1. Perform identification and authorization section 6.1 “Identification and Authentication” on page 6-1 (Step 1)
2. Check authorization to device description (*DEVD) for display station (Step 2)
3. Check authority to associated objects (job description, output queue, print device, job queue, current library, libraries on library list) (Step 2)

The subsystem monitor is responsible for assigning the permanent job structure (WCBTE), allocating the temporary job structure (WCB), and initializing the PDT. It also initiates the user process.

After the MI instruction Initiate Process, the process starts on behalf of the user and the initiation and problem phase programs run within that user process.

Interactive jobs are ended as a result of the following activities:

- User choosing sign off option from the main menu
- The CL command Sign Off (SIGNOFF)
- The CL command End Subsystem (ENDSBS) to end the subsystem in which the job is running
- The CL command Power Down System (PWRDWNSYS) to power the system down
- Powering down the display station
- Canceling a job as part of the work with jobs CL command End Job (ENDJOB)
- After a designated period of no activity as specified by a system value
- The CL command End System (ENDSYS)

In addition, a system value is used to determine what action should be taken when an I/O error occurs on the device. The possible values include ending the user’s session. When a user’s session is terminated, all other jobs attached to *DEVD object are ended.

Using the CL commands ENDSBS, ENDSYS, and PWRDWNSYS require *ALLOBJ and *JOBCTL special authority.

Once an interactive job is ended on the system, the subsystem monitor job displays the sign-on screen on the display station. The operating system code that runs during process termination signals an event to the subsystem monitor job indicating that the user job has ended. The subsystem monitor operating system code then displays the sign-on screen on the device that has been transferred back to the subsystem monitor job.

4.1.3.3.4. Batch Processing Jobs

The primary difference between the interactive support on the AS/400 system and the support for batch jobs is that batch functions are processed as a result of entries placed on a job queue instead of signing onto the system.

A job queue entry in the subsystem description specifies that jobs can be started using the job queue. When the batch subsystem is started, it processes the jobs on the queue. A subsystem description can specify the maximum number of jobs that can be processed at the same time.

Not all jobs on a job queue are necessarily available for processing when the subsystem is started; jobs can be held on a queue until the system operator releases them.[5] If the subsystem is ended before all the jobs are

[5] The queue must be empty before it is deleted.

processed, the jobs remain on the queue until the subsystem is started again or until another subsystem allocates the same job queue. More than one subsystem description can refer to the same job queue, but only one active subsystem at a time can use the job queue as a source of batch jobs.

Figure 4-8. “Batch Job Processing” on page 4-34 illustrates the steps involved in processing a batch job.

There are CL commands to place a job on a job queue.

A batch job on any job queue can be moved to a different queue. A CL command is used to change an existing job queue entry while the subsystem is active. An active subsystem is one that has been started, for example, with the CL command Start Subsystem (STRSBS); an inactive subsystem is one that has ended, for example, with the CL command End Subsystem (ENDSBS) or has not been started. To change job queues, the following authorities are needed:

- The user profile associated with the process must match the user profile associated with the job to be matched; the user needs *USE to the new Job Queue
- If the user profiles do not match, then the user also needs Job Control (*JOBCTL) special authority

Parameters for batch jobs come from the following sources:

- As parameters on the command
- Job description specified in the command
- Current user’s user profile
- Currently active job (job issuing the CL command Submit Job (SBMJOB))

Submitting a batch job requires *READ authority to the job queue.

As with interactive jobs, batch jobs can specify in the job description to use the user profile. Once the job has all of its attributes, it is placed on the job queue by the Spool component by request of the Submit Job Command Processing Program (CPP).

When the subsystem is ready to handle a job, it looks in the job queues (those that the subsystem has allocated), then checks the job description for routing data. After this information is obtained, the routing program is run. The job description used for batch jobs is specified on the CL commands Submit Batch Job or Batch Job (BCHJOB).

If an input stream is entered that contains the CL command BCHJOB, the user entering the CL command Start Diskette Reader, Start Database Reader, Submit Diskette Job, or Submit Database JOB must have the appropriate authority to the specified job description. If USER (required) is specified in the job description, it is not valid to use the job description on a BCHJOB command. The CL command BCHJOB does not have a parameter for user profile.

If a CL command SBMJOB is used, the command defaults so that the batch job operates under the user profile name of the submitter. However, if USER(*JOBD) is specified on the SBMJOB command, the job operates under the name in the job description. The submitter needs *USE authority to the job description to run under the profile in the job description and *USE authority to the User Profile.

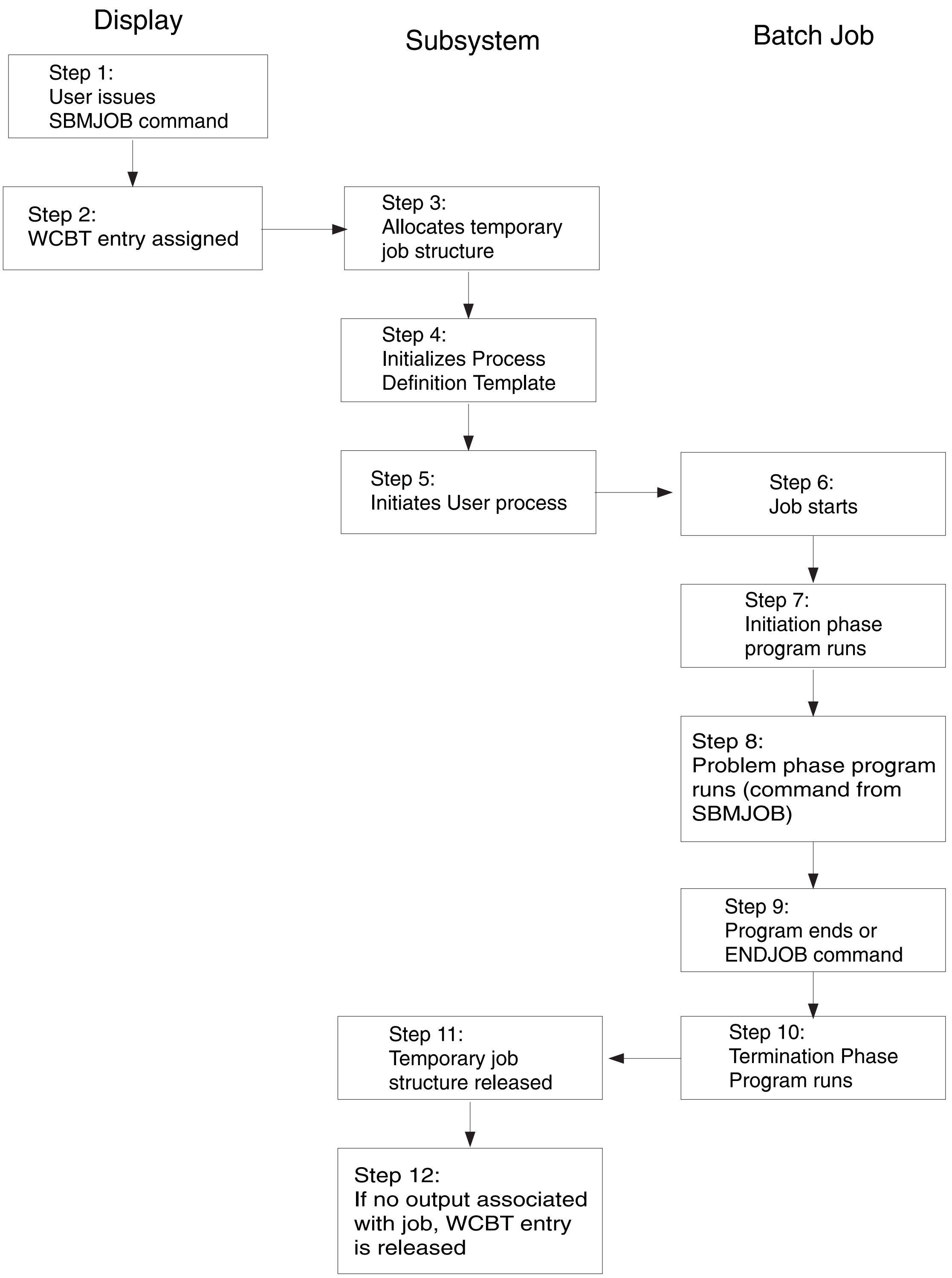


Figure 4-8. Batch Job Processing

When the CL command Submit Job is issued, it performs the following functions:

- Sets up parameters for Job function
- Invokes the CPP for the command to verify the command issuer’s authority to the Job Description (JOBD), verify the job user’s authority to the JOBD, verify the job attributes, assign the permanent job structure, and initialize the spool control block, job message queue, and WCBT entry
- Sets up the display submitted job field in the job’s job structure of the user for whom the job was submitted
- Calls Spool to enqueue a job request to the Job Queue
- Returns the job name in the job log
- Sends the message or return status of the submission to the invoker

The CPP of the CL command Submit Job checks for authority to the user profile and updates the WCBTE with the name of the user profile under which the job is to run. A pointer to the user profile is stored in the WCB and does not become associated with the job until the job becomes active. The temporary job structure (WCB) is created as part of process initiation. Authority to the Job Queue is checked before calling Spool.

When the job attempts to start, the corresponding process initiation activities perform authority checks, including checking for the presence and authority to the user profile. An exception results in the process not initiating and the job not starting.

Parameters on the CL command Submit Job (SCDDATE and SCDTIME) allow for scheduling when a batch job is allowed to run.

4.1.3.4. Spool

The Simultaneous Peripheral Operations On-Line (Spool) Component provides support for submitting batch jobs onto the system and for managing the spooled printing of output data. The Spool Maintenance system job, which performs system spool functions, is also part of the Spool Component. The support provided by Spool for submission of batch jobs is described in section 4.1.3.4.3 “Batch Support and Spooling Readers” on page 4-36, and the support for spooled printing is described in section 4.1.3.4.4 “Spooled Print Support and Spooling Writers” on page 4-40.

To perform its functions, Spool maintains a set of database physical files in the QSPL library. Spool also uses data structures related to the user’s job on whose behalf it is performing actions. The data structures used by Spool and their use is described in the appropriate sections that follow.

4.1.3.4.1. Spool Component Database Physical Files

Database physical files, which are managed by the AS/400 Database System (see section 4.2 “Database Management System” on page 4-68), are used by Spool to hold input data for batch jobs and spooled output print data.

The physical files used by the Spool Component are contained in the QSPL library, which is owned by the user profile QSPL and is shipped with *EXCLUDE public authority.[6] The physical files are also owned by user profile QSPL. Their public authority is *CHANGE; however, they are protected from access by anyone other than QSPL as long as public authority to the QSPL library is *EXCLUDE.

[6] The TFM warns administrators that changing public authority on the QSPL library or giving untrusted users private authorities to this library violates the evaluated configuration.

Most Spool Component programs are owned by user profile QSPL and have the adopt owner authority bit set. In this way, these programs can access the Spool physical files.

At installation, Spool creates one physical file for each defined record length (for example, for card image input data, a physical file is created with record length of 80 bytes; another is created with record length of 4 KB for spooled print data). Spool uses a physical file for each record length until the physical file contains approximately 1000 members; Spool creates additional physical files for that record length as required.

These physical files are permanent objects in the system, and the database members within these physical files are reused to hold batch input data and print data for different jobs.

Since database members are reused, Spool clears each database member before using it for another job's input data or print data. See section 4.1.3.4.3.4 "Creating Inline Data Files" on page 4-38 and section 4.1.3.4.4.2 "Spooled Files" on page 4-41 for more details.

Spool maintains a File Availability Control Block (FACB) to keep track of all spool database physical files and members. There is one FACB on the system, and it contains an entry for each spool database physical file and an entry for each database file member.

4.1.3.4.2. Spool Data Structures

As described in section 4.1.3.3.1 "Job Structure" on page 4-24, every job has an entry in the WCBT called the *WCBT Entry*, and each WCBT Entry points to a Spool Control Block (SCB) data structure. In addition, while a job is active, it has associated with it a data structure called the WCB. The WCB also points to the SCB data structure.

The SCB contains an Inline File Control Block (INFCB) for every inline data file used by the job (batch jobs only) and an Output File Control Block (OFCB) for every spooled file created by the job.

The INFCB points to the database member containing the inline data.

The OFCB points to the database file member and to the output queue that the spooled file resides on. It also contains all the attributes for the spooled file with which it is associated.

4.1.3.4.3. Batch Support and Spooling Readers

Spool provides the following support for batch jobs:

- Submitting batch jobs either using a CL command or by starting a spooling reader
- CL commands to list spooling readers and to perform operations on spooling readers such as starting, holding, releasing, and ending
- Creating inline data files
- CL commands to perform operations on job queues such as creating, holding, releasing, deleting, and working with job queues

Each of these activities is described in more detail below. Additionally, as background for understanding the information presented in the following sections, section 4.1.3.4.3.1 "Batch Job Input Stream" on page 4-36 provides an overview of the contents of a batch job input stream.

4.1.3.4.3.1. Batch Job Input Stream

A batch job input stream is a card image input stream that can exist either in a database file or on a diskette device. It consists of multiple card images.

4.1.3.4.3.2. Submitting Batch Jobs Via CL Commands

Spool provides support for submitting (i.e., “spooling”) batch jobs into the system. As Spool accepts a batch job into the system, it calls a Work Management Component, which adds the batch job to the appropriate job queue. The Work Management Component subsequently takes each batch job entry off the job queue and creates a batch job, as described in section 4.1.3.3 “Job Management” on page 4-23

The CL commands provided by Spool are described below, and spooling readers are described in section 4.1.3.4.3.3 “Spooling Readers” on page 4-37.

Spool provides two CL commands to submit batch jobs. The CL command Submit Database Job (SBMDBJOB) submits a batch job from a database file.[7] The authorities required to submit a batch job using the CL command SBMDBJOB are shown in Table 4-2 on page 4-37. The CL command Submit Diskette Job (SBMDKTJOB) submits a batch job from a diskette file. The authorities required to submit a batch job using the CL command SBMDKTJOB are shown in Table 4-3 on page 4-37. To successfully submit a job, additional authority checks are performed against other objects by Job Management (e.g., user profile, *JOBD, *OUTQ, etc.). These checks are discussed in section 4.1.3.3.4 “Batch Processing Jobs” on page 4-32.

Table 4-2. Required Authorities for SBMDBJOB

Object Type	Required Authority	Notes
Job Queue (*JOBQ)	*READ	
Database File (*FILE)	*USE	
Library (*LIB)	*EXECUTE	Libraries that contain the Job Queue and Database File

Table 4-3. Required Authorities for SBMDKTJOB

Object Type	Required Authority	Notes
Job Queue (*JOBQ)	*READ	
Message Queue (*MSGQ)	*OBJOPR and *ADD	Message Queue to receive operator messages such as “Diskette Read Error”
Device Description (*DEVD)	*READ	
Library (*LIB)	*EXECUTE	Libraries that contain the Job Queue and Message Queue

4.1.3.4.3.3. Spooling Readers

Spooling readers are system jobs that support submitting batch jobs on the system. They are started and manipulated via CL commands.

There are two types of spooling readers: *database readers* and *diskette readers*. A database reader reads a batch input stream from a database file member and places it on the job queue specified in the CL command

[7] For clarification, the database file referred to here is a database file member containing a batch job input stream; it is not one of the database files in the QSPL library that Spool uses internally.

that started the spooling reader job. A diskette reader reads a batch job from a diskette device and places it on the specified job queue. Spooling readers run as jobs under the user profile QSPLJOB user profile.

The Spool Component maintains a data structure called the *Reader/Writer Control Block*, which contains an entry for every reader and writer job running on the system. One of the fields is the user profile of the user who started that reader/writer job. Since readers run under the user profile QSPLJOB user profile, before calling the Work Management program to submit a batch job, the reader “swaps” to the user profile of the user who started it. In this way, each batch job that the reader submits to Work Management is submitted under the user profile of the user who started the reader.

The authorities required to start a spooling reader are shown in Table 4-4 on page 4-38.

Other CL commands that manipulate spooling readers perform the following operations:

- End a reader - Terminates the reader job.
- Hold a reader - Causes reader to immediately stop reading data.
- Release a reader - Allows reader to resume reading data.

To end, hold, or release a spooling reader, the user attempting the operation must do one of the following:

- Be the user who originally started the reader (CPP associated with the CL command checks the Reader/Writer Control Block to determine who started the reader job)
- Must have *ALLOBJ or *JOBCTL special authority (CPP associated with the CL checks for special authority)

Table 4-4. Authorities Required to Start Spooling Reader

Object Type	Required Authority	Notes
Message Queue (*MSGQ)	*OBJOPR and *ADD	Message Queue to receive operator messages such as “Reader Job Started” and “Diskette Read Error”
Job Queue (*JOBQ)	*READ	
Database File (*FILE)	*USE	Database Reader Only
Device Description (*DEVD)	*READ	Diskette Reader Only
Library (*LIB)	*EXECUTE	Libraries that contain the Message Queue, Job Description, and Database File.

4.1.3.4.3.4. Creating Inline Data Files

As discussed previously, the CL command SBMDBJOB, the CL command SBMDKTJOB, and Spooling Readers interpret batch input streams from either database files or diskette devices and place jobs on job queues. When the batch input stream contains *inline data* (e.g., source statements for a compiler, input data for an application program), Spool copies the inline data to a member of its database file with the appropriate record length.

When the batch job subsequently runs, it calls a Spool Component program (*PGM) to set up the DMCQ with information that permits the Data Management Component programs to open and read the job’s input from the Spool database file member. The *PGM adopts user profile QSPL authority (i.e., owner of *PGM object) so that it can access the database file member on behalf of the batch job (since it is protected by DAC from access by other than user profile QSPL).

As described in section 4.1.3.3.1 "Job Structure" on page 4-24, every job has a WCBT entry associated with it that points to the SCB data structure. The SCB contains an INFCB for every inline data file needed by that batch job. The INFCB points to the database member containing the inline data.

4.1.3.4.3.5. Job Queues

Job queues (*JOBQ objects) are used to hold batch jobs until they are started by a batch subsystem, as described in section 4.1.3.3.4 "Batch Processing Jobs" on page 4-32. They contain one entry for each batch job that has been submitted for execution. Spool is the object manager for job queue objects. Via CL commands, Spool provides support to manipulate job queues, as described in this section. Spool provides CL commands to perform the following operations on job queues:[8]

- Create a job queue
- Delete a job queue
- Hold a job queue
- Release a job queue
- Clear a job queue
- Work with a job queue - This command lists all jobs on a specified job queue. The information displayed includes the qualified job name, date submitted, and time submitted.

When Spool creates a job queue, it automatically gives the IBM-supplied user profile QSPL *CHANGE private authority to the job queue. Since most Spool programs (*PGM) are owned by user profile QSPL and run with user profile QSPL adopted authority, they have authority to each job queue. Therefore, Spool must explicitly check the user's authority to a job queue before manipulating the job queue on behalf of a user. A summary of the policy enforced by Spool on job queues follows:

- A user with *SPLCTL special authority can clear, hold, release, and work with any job queue in all libraries that the user has *EXECUTE authority to.
- A user must have *OBJEXIST authority to a job queue to delete it.
- The Create Job Queue (the CL command CRTJOBQ) has parameters that control how authority checking is subsequently done on the job queue. These parameters are described below:

Operator Control (OPRCTL) Parameter- specifies whether or not a user with *JOBCTL special authority is allowed to control the job queue being created (i.e., clear, hold, release, and work with job queue) and to manage jobs on that job queue. The OPRCTL parameter has one of the following values:

***YES**: User with *JOBCTL can clear, hold, release, and work with that job queue.

***NO**: *JOBCTL special authority is not honored by Spool for that job queue.

[8] CL commands are also available to manipulate the jobs on job queues; these commands are provided by the Job Management Component and described in section 4.1.3.3.4 "Batch Processing Jobs" on page 4-32.

Authority to Check (AUTCHK) Parameter- specifies whether or not the clear, hold, release, and work with job queue CL commands require ownership on that job queue to perform the associated operation.[9] The AUTCHK parameter has one of the following values:

***OWNER**: Clear, hold, release, and work with job queue CL commands require ownership.

***DTAAUT**: Clear, hold, release, and work with job queue CL commands require *READ, *ADD, and *DELETE authorities on the job queue.

4.1.3.4.4. Spooled Print Support and Spooling Writers

On AS/400, an application job can either send print output data directly to an output device (e.g., printer or diskette device) as the print data is generated or the job can send (i.e., "spool") the print data to a *spooled file*. The spooled file remains on an *output queue* (*OUTQ object) until the data in the spooled file is sent to an output device by a *spooling writer* independently of the job running. To send data directly to an output device, the user profile associated with the application job must have *READ authority to the *DEVD object that represents the printer. The Spool Component is not involved when output data is sent directly to an output device. For spooled printing, the Spool Component provides the following support:

- Interception of print attributes and print data to create spooled files
- CL commands to perform operations on output queues and on spooled files in output queues
- Spooling writer jobs that send print data to an output device (*DEVD object type)
- CL commands to manipulate spooling writers

Each of these activities is described in more detail in the sections that follow.

4.1.3.4.4.1. Printer Files

A printer file is an object of type *FILE with attribute PRTF. Printer files are created by users using the CL command Create Printer File (CRTPRTF) independently of running application jobs, and contain print attributes that control the printing of the print data. Examples of these attributes are page size, lines per inch, and font. Other attributes contained in printer files include the following:

- An attribute that controls whether the print data is spooled or printed immediately.
- When print data is to be spooled, it also contains the name of the output queue to hold the spooled file until it is printed.
- An attribute that controls whether the spooled file is released for re-use immediately after it is printed or held until it is explicitly deleted by a CL command.

On AS/400, application jobs cause output data to be printed by "writing" to a printer file. Note that the print data is not actually "written" to the printer file. Instead, the printer file determines the print attributes and whether the print data is spooled or printed directly.

[9] Users with *SPLCTL special authority or with *JOBCTL special authority (when *OPRCTL is *YES) *and have *EXECUTE authority to the library* need neither ownership nor any authority to the job queue to perform these operations.

4.1.3.4.4.2. Spooled Files

Whenever print data is to be spooled, a Spool program is called. This program performs the actions described below and illustrated in section 4-9. “Adding Data to Spooled File” on page 4-41.

- Through the FACB, Spool selects an empty database member (or creates one if an empty one is not available).
- It updates the OFCB to point to the database file member.
- It checks that the user profile associated with the job that invoked Spool has at least *READ authority to the output queue. If so, it updates the OFCB to point to the output queue.
- It updates the appropriate output queue to contain the print control information and to point to the database file member, the OFCB, and to the WCBT entry. It is through the WCBT entry that the “owner” of the spooled file can be found.

After the application job completes, the spooled file is sent to an output device (e.g., a printer) by a *spooling writer*, as shown in Figure 4-10. “Printing of Spooled File by Spool Writer” on page 4-42. (See section 4.1.3.4.4.4 “Spooling Writers” on page 4-45).

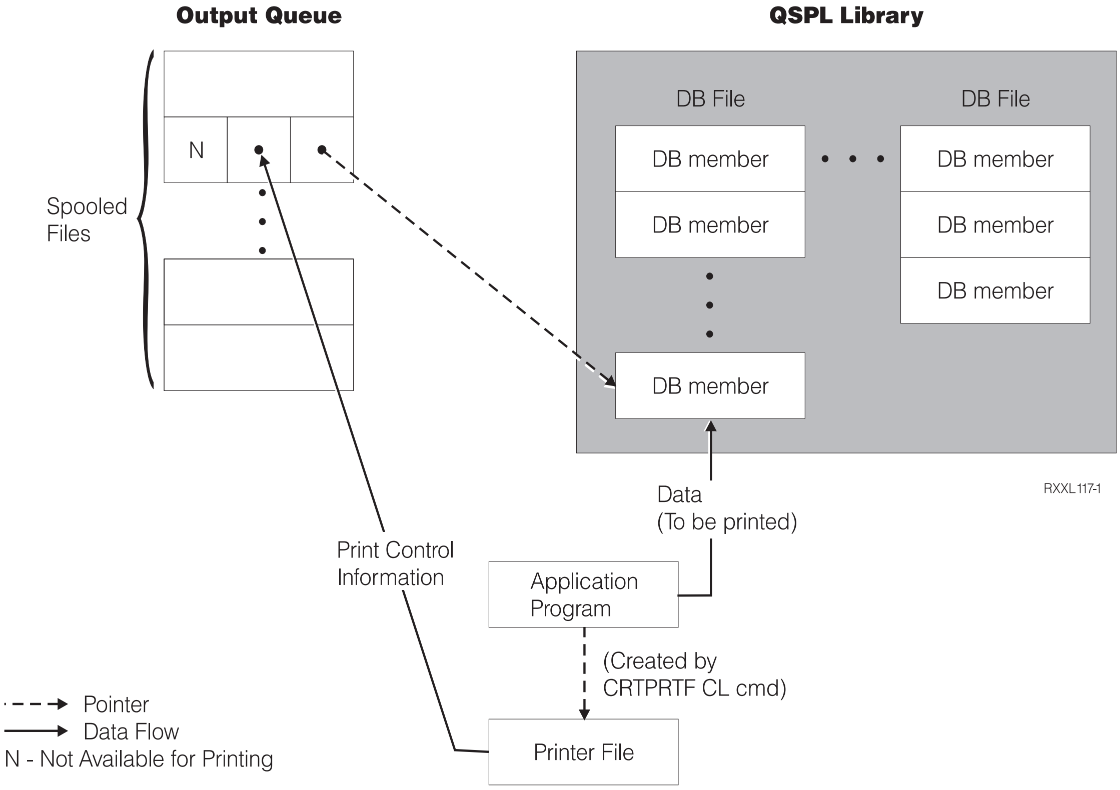


Figure 4-9. Adding Data to Spooled File

4.1.3.4.4.3. Output Queues

Output queues (*OUTQ objects) are used to hold spooled files for printing. They contain one entry for each spooled file that exists on the queue. Each spooled file entry contains the following information:

- Availability code
- Available for printing (indicated by “A” in Figure 4-10. “Printing of Spooled File by Spool Writer” on page 4-42)
- Not available for printing (indicated by “N” in Figure 4-9. “Adding Data to Spooled File” on page 4-41)
- Pointer to WCBTE
- Pointer to OFCB
- Pointer to database file member containing print data
- Print Control information

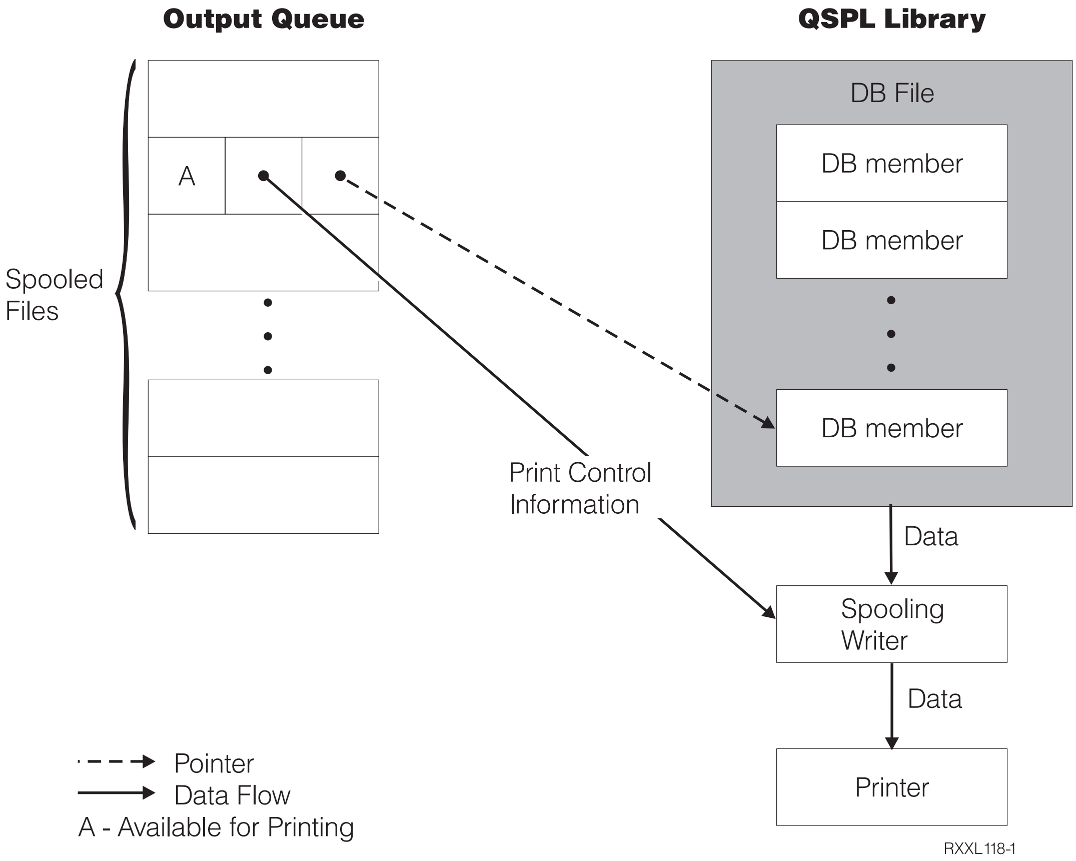


Figure 4-10. Printing of Spooled File by Spool Writer

Spool is the object manager for output queues and for spooled files. Via CL commands, Spool provides support to manipulate output queues and their associated spooled files. The list below summarizes the operations that can be performed on output queues:

- Create an output queue.
- Change the attributes of an output queue.
- Clear an output queue. This CL command deletes all spooled files in the specified output queue that are waiting to be printed. Files currently printing are not deleted.
- Delete an output queue.
- Hold an output queue. Prevents all spooled files, on the output queue, from printing until the output queue is “released” via the CL command Release Output Queue.
- Release an output queue.
- Start a spooling writer on an output queue.
- Work with an output queue.

The list below summarizes the operations that can be performed on spooled files in an output queue:

- Display a list of spooled files on an output queue. The information displayed includes the following:
    - Spooled file name (obtained from the printer file)
    - Spooled file number (unique number within a job)
    - Qualified job name
    - Date and time submitted
- Display the contents of a spooled file on an output queue.
- Move a spooled file from one output queue to another.
- Copy a spooled file to a source physical file (*FILE).
- Delete a spooled file.
- Hold a spooled file.
- Release a spooled file.
- Change the print attributes associated with a spooled file.

When an output queue is created, security parameters can be specified to control the functions that users can perform on the output queue and on the spooled files within the output queue. These parameters are described below:

Operator Control (OPRCTL) Parameter- specifies whether or not a user with *JOBCTL special authority is allowed to control the output queue being created and to manage spooled files on that output queue without having specific authorities to the output queue.

***YES**– *JOBCTL special authority is honored by Spool for performing certain operations; no authority to that output queue is required.

***NO**– *JOBCTL special authority is not honored by Spool for that output queue.

Authority to Check (AUTCHK) Parameter- specifies whether or not ownership on the output queue is required to perform the associated operation.[10]

***OWNER**– Clear, hold, release, and work with output queue CL commands require ownership.

***DTAAUT**– Clear, hold, release, and work with output queue CL commands require *READ, *ADD, and *DLT authority to the output queue.

Display Data (DSPDTA) Parameter- determines authority required to perform the following functions on spooled files owned by other users: display spooled file contents, copy spooled file, and move spooled file.

***NO**– Users can display, copy, and move only spooled files on the queue that they own unless the user has one of the following:

*SPLCTL special authority

*JOBCTL special authority (if OPRCTL is *YES)

*READ, *ADD, and *DLT authority to the output queue (if *AUTCHK is *DTAAUT)

Ownership of output queue (if *AUTCHK is *OWNER)

***YES**– Users with *READ authority to the output queue can display, copy, and move all spooled files on the output queue.

***OWNER**– Users can display, copy, and move only spooled files on the queue that they own. Users with *JOBCTL special authority cannot display, copy, or move spooled files. However, they can hold, change, delete, and release spooled files (if *OPRCTL is *YES). This is intended to allow operators to manage entries on an output queue without being able to view the contents.

When Spool creates an output queue, it automatically gives the user profile QSPL *CHANGE private authority to the output queue. Since most Spool programs (*PGM) are owned by user profile QSPL and run with user profile QSPL adopted authority, they have authority to each output queue and the spooled files on that output queue. Therefore, Spool must explicitly check the user's authority to an output queue before manipulating the output queue on behalf of the user. The checks performed by Spool are summarized below and in Table 4-5 on page 4-45.

- A user with *SPLCTL special authority is not subject to any authority restrictions associated with output queues or with the spooled files in output queues. A user with *SPLCTL special authority and *EXECUTE authority to each library the output queues are in can perform all operations on all output queues.
- A user must have *OBJEXIST authority to a output queue to delete it.
- A user must have *READ authority to an output queue to add a spooled file to the output queue.
- Users can always display and manipulate any spooled files that they own in an output queue, regardless of their authority to the output queue holding the spooled files.

[10] Users with *SPLCTL special authority or with *JOBCTL special authority (when *OPRCTL is *YES) *and *EXECUTE authority to the library* do not need ownership nor any authority to the output queue to perform these operations.

4.1.3.4.4.4. Spooling Writers

A *spooling writer* is a job that monitors an output queue and sends spooled files from that output queue to an output device. There are two types of spooling writers: printer writers and diskette writers. Printer writers send print data to a printer, and diskette writers send print data to a diskette device. Spooling writers run as jobs under the user profile QSPLJOB user profile, and they are started and manipulated via CL commands.

Depending on the value of a printer file option, the spooling writer may release the database member for reuse as soon as the data is sent to an output device. In this case, the spooling writer clears the data in the database member before releasing the member.

Table 4-5. Operations on Output Queues and Authorities Required

Function	DSPDTA	AUTCHK	OPRCTL	Required Output Queue Authority	Special Authority
Add Spooled File to Queue	Any	Any	Any	*Read	None
	Any	Any	*YES	None	*JOBCTL
	Any	Any	Any	None	*SPLCTL
View List of Spooled files	Any	Any	Any	*READ	None
	Any	Any	*YES	None	*JOBCTL
	Any	Any	Any	None	*SPLCTL
Display, Copy, or Move Spooled Files[a]	*YES	Any	Any	*READ	None
	*NO	*DTAAUT	Any	*READ,*ADD, and *DLT	None
	*NO	*OWNER	Any	Owner	None
	*OWNER[b]	Any	Any	None	None
	*YES/*NO	Any	*YES	None	*JOBCTL
	Any	Any	Any	None	*SPLCTL
Change, Delete, Hold, and Release Spooled File[c]	Any	*DTAAUT	Any	*CHANGE	None
	Any	*OWNER	Any	Owner	None
	Any	Any	*YES	None	*JOBCTL
	Any	Any	Any	None	*SPLCTL
Change, Clear Hold, and Release Output Queue	Any	*DTAAUT	Any	*CHANGE[d]	None
	Any	*OWNER	Any	Owner[d]	None
	Any	Any	*YES	None	*JOBCTL
	Any	Any	Any	None	*SPLCTL
Delete Output Queue	Any	Any	Any	*OBJEXIST[d]	None
	Any	Any	*YES	*OBJEXIST[d]	*JOBCTL
	Any	ANy	Any	*OBJEXIST[d]	*SPLCTL

[a] Spooled file owners can display, copy, and move their spooled files regardless of authority of source output queue. Move operation requires *READ authority to destination output queue. Copy operation requires *USE authority to destination source physical file.

[b] Only the owner or user with *SPLCTL special authority can display, copy, or move spooled file when DSPDTA is *OWNER.

[c] Spooled file owners can change, delete, hold, and release their spooled files regardless of their authority to output queue.

[d] Must have *EXECUTE authority to the library the output queue is in.

Table 4-6. Operations on Spooling Writers and Authorities Required

Function	AUTCHK	OPRCTL	Required Output Queue Authority	Special Authority
Start Writer[a]	*DTAAUT	Any	*READ, *ADD, *DLT	None
	*OWNER	Any	Owner	None
	Any	*YES	None	*JOBCTL
	Any	Any	None	*SPLCTL
Change Writer	Any	Any	*READ, *ADD, *DLT[b]	None
	Any	*YES	None	*JOBCTL
	Any	Any	None	*SPLCTL
End, Hold, Release Writer	Any	Any	*READ, *ADD, *DLT	None
	Any	*YES	None	*JOBCTL
	Any	Any	None	*SPLCTL

[a] Requires *OBJOPR and *ADD to message queue, *EXECUTE to the message queue library and *USE to printer device description (*DEVD).

[b] Authority required to new output queue; *EXECUTE authority is required to the library the new output queue is in; no authority is required to the old output queue.

Alternatively, a spooled file (and the associated database member) may continue to exist until it is explicitly deleted by the user. In this case (since a spooling writer may not be active for the output queue in which the spooled file resides), there is a system job **QSPLMAINT** (see section 4.1.3.4.5 “Spool Maintenance System Job” on page 4-47 that is responsible for clearing database members that are released independently of a spooling writer.

A spooling writer is associated with only one output queue and one output device. Until a spooling writer is running for an output queue, the spooled file is not sent to the output device. The spooling writer associated with a particular output queue periodically checks that queue for spooled files that are ready to print (a field in the output queue for the spooled file is set to “A” when the spooled file is available to be printed, as indicated by “A” in Figure 4-10. “Printing of Spooled File by Spool Writer” on page 4-42).

The following CL commands are available to start and manipulate spooling writers. The authorities required to perform these operations are dependent on the security parameters of the output queue and are summarized in Table 4-6 on page 4-46.

- Start a writer - Starts writer job for specified output queue and specified output device.
- Change a writer - Changes the output queue from which the writer accepts spooled files.
- End a writer - Terminates the writer job.
- Hold a writer - Causes writer to stop writing data.
- Release a writer - Allows writer to resume writing data.

4.1.3.4.5. Spool Maintenance System Job

The Spool Maintenance system job performs system spool functions. It is started as part of the IPL and runs under the user profile QSYS. It performs the following actions:

- Zeros the contents of spool database members after a user has deleted a spooled file (see section 4.1.3.4.4.2 "Spooled Files" on page 4-41).
- Deletes spool database members that are unused and empty to free up virtual address space. A system value controls the number of days that unused spool database members are held before they are deleted.
- Performs cleanup whenever a deleted spool database file is encountered. QSPLMAINT removes the spooled file (i.e., pointers to deleted database file members) associated with the deleted spool database file.
- QSPLMAINT also performs actions as part of an abnormal IPL.[11]
  - Moves batch jobs from destroyed job queues to the QSPRCLJOBQ job queue in the QRCL library.
  - Moves spooled files from destroyed output queues to the QSPRCLOUTQ output queue in the QRCL library.
  - Moves spooled files from destroyed output queues to newly created output queues.

4.1.3.5. Command Analyzer

The Command Analyzer (CA) processes all CL commands (*CMD objects). Command processing consists of parsing the command, validity checking each command parameter, and transferring control to a Command Processing Program (CPP) (*PGM object).

CA uses information contained in the *CMD object to perform validity checking[12] and to determine the format of parameters to be passed to the CPP. The *CMD object also contains the following information:

- Name of an optional Validity Checking Program (VCP)
- Offset of an optional VCP in EPT (for IBM-supplied CL commands with a VCP in the EPT)

[11] An "abnormal IPL" is an IPL after an abnormal shutdown (e.g., power failure).

[12] The validity checking performed by CA are non-security relevant checks, such as maximum length of a character string, and bounds and range checking.

- Offset of CPP in EPT (for IBM-supplied CL command with a CPP in the EPT)
- Name of the CPP that processes the command
- Name of an optional Prompt Override Program (POP)
- Offset of POP in EPT (for IBM-supplied CL commands with a POP in the EPT)
- Information used by CA to determine whether to call each of the above programs (VCP, POP, and CPP) in system or user state. Only programs for IBM-supplied commands can be called in system state.
- Information used by CA to determine whether the CL command is an IBM-supplied CL command or a user-created command.
- Names of optional prompt control programs (PCP)
- Offsets of PCPs in EPT (for IBM-supplied commands with PCPs in the EPT)
- Names of optional prompt choices programs
- Offsets of prompt choices programs in EPT (for IBM-supplied commands with prompt choices programs)

CA is called by all components that interpret CL commands. Some use it only to parse and validity check commands. In this case, CA places the parsed and validity checked form of the command in a *positional list*, which is then passed back to the caller to be used when the command is executed at a later time. Some components that use CA in this manner include:

- CL compiler (as the commands are being compiled)
- The CL command Create Command (as a command is being created)
- Spooling (for job commands)
- Parameter Prompter
- Source Entry Utility (SEU)

Other components call CA to parse, validity check, and immediately execute the command. These include the following:

- QCMD (Message Handler Component)
- Interpretive command entry processor
- CL programs
- Prompter (to execute a command after it is completely entered and processed)
- High-Level Language (HLL) Programs (to process commands using the UDSS interface provided by CA)

CA provides two interfaces: UDSS programs that can be used by user programs to invoke CA and SDSS programs used from within OS/400.

The subsections that follow describe the MI space objects and MI queue objects used by CA and its calling components to exchange command strings and positional lists, the macros that manipulate the space and queue objects, the command processing performed by CA, and the support provided by CA during the deletion of a user profile.

4.1.3.5.1. CA Spaces and CA Queue

To process a command, CA uses two MI space objects. Collectively, these objects are referred to as *protected CA spaces*. These two objects are described below:

- **Positional List (PL)**: The PL is a system domain MI space object. The PL contains a pointer to the Work Area (WA) (described below). CA places the parsed and validity checked form of the command in the PL.
- **Work Area (WA)**: The WA is a write-protected user domain space object. When CA is called, the WA contains the command string on which CA will operate.

Since CA spaces are used by components that call CA, CA maintains an MI queue object that contains the CA spaces. A separate queue is created for each user profile on the system as it is needed, and the user profile associated space contains a pointer to this queue. This allows the CA spaces to be reused by all jobs that run under the same user profile. CA provides a set of macros to enqueue and dequeue CA spaces to and from the CA queue. If the user profile associated with a job has changed, the enqueue macro does not return freed-up CA spaces back to the user’s queue. Instead, it destroys the CA spaces to prevents jobs associated with a different user profile from gaining information about another user’s job.

The queue and the MI space objects are created as temporary objects. Consequently, their storage is freed up at IPL time.

Under certain circumstances, user state programs need access to the PL and WA. Consequently, the macros support an option to create a user domain non-write-protected PL and WA, which are collectively referred to as unprotected CA Spaces. CA manages the unprotected CA Spaces identically to the protected CA Spaces.

4.1.3.5.2. Command Processing Overview

An overview of the command processing performed by CA is described below:

- CA is called with a system pointer to the PL, which points to the WA, which contains the command string to analyze. CA first parses the command string, creating a token list from the various parameters (command name, keyword, keyword value, etc.) of the command. The token list is placed in the WA.
- CA resolves the command name to the *CMD object.
- The authority to the *CMD object is checked by the MI instruction.
- CA uses information in the *CMD object to validity check each parameter and build a positional list in the PL. If a Validity Checking Program (VCP) is specified in the *CMD object, CA calls it to perform extended validity checking. CA invokes the MI instruction Call External (CALLX) with the drop state option unless the *CMD object indicates that the command is IBM-supplied and that the VCP should be called in system state.
- If the command is to be executed, CA invokes the MI instruction Transfer Control (XCTL) to call the appropriate CPP, passing to it pointers to each of the keyword values in the PL. CA invokes XCTL with the drop state option, unless the *CMD object indicates that the command is IBM-supplied and that the CPP should be called in system state. This prevents a user-created CL command from calling a System Domain CPP (*PGM object).
- If the command is not to be executed, CA returns the positional list to its caller.

The command processing performed by CA and the path through CA may be slightly different depending on how it is called. These paths are described below:

- **CL command issued interactively**: A system state program gets a set of protected CA spaces using the CA dequeue macro. It places the command string in the WA. It then invokes CA through the SDSS interface, QCADRV2. If the CPP is user state, CA copies the parameters to be passed to the CPP to unprotected CA spaces before calling the CPP.[13]
- **CL command execution from a IBM-created (System State) CL Program**: Works as described above.
- **CL command execution from a user-created (User State) CL Program**: The CL program calls QCADRV, which is UDSS. QCADRV gets a set of protected CA spaces, copies parameters from the CL program's associated space to the protected CA spaces, and performs parameter validation (as described in section 4.1.2.2.4 "User Domain System State Programs" on page 4-16). It then continues processing as described above.
- **Command Analyzer APIs**: CA has three APIs, which are UDSS, and, as such, performs parameter validation when invoked as described in section 4.1.2.2.4 "User Domain System State Programs" on page 4-16.
  - QCMDCHK: The QCMDCHK API is used to parse and validity check a command string. A High Level Language (HLL) user program calls QCMDCHK, passing it a command string to process. QCMDCHK allocates protected CA spaces, copies the command string from user program storage to the protected WA, and performs parameter validation (as described in section 4.1.2.2.4 "User Domain System State Programs" on page 4-16). CA parses the command string to create a positional list, which it then copies back to user program storage. QCMDCHK deallocates the CA spaces, and then returns control to the HLL program.
  - QCMDEXC: The QCMDEXC API is used to parse, validity check, and then immediately execute a CL command. An HLL user program calls QCMDEXC, passing it a command string to process. QCMDEXC allocates protected CA spaces, copies the command string from user program storage to the protected WA, and performs parameter validation (as described in section 4.1.2.2.4 "User Domain System State Programs" on page 4-16). CA parses the command string to create a positional list. It then calls the CPP as previously described. On return from the CPP, QCMDEXC deallocates the CA spaces, and then returns control to the HLL program.
  - QCAPCMD: The QCAPCMD API is used to parse, validity check, and optionally execute a CL command. It combines the functions of QCMDCHK and QCMDEXC and provides some other CA functions, such as displaying the help for a command. An HLL user program calls QCAPCMD passing it a command string to process, along with parameters that tell CA what type of processing to perform (for example, whether the command should be executed or simply parsed and validity checked). QCAPCMD allocates protected CA spaces, copies the command string from user program storage to the protected WA, and performs parameter validation (as described in section 4.1.2.2.4 "User Domain System State Programs" on page 4-16). CA parses the command string to create a positional list. If the command is to be executed, CA then calls the CPP as previously described. On return from the CPP, QCAPCMD deallocates the CA spaces, and then returns control to the HLL program.

[13] The same thing is done before calling a user state VCP, and if a POP, PCP, or choices program is user state, the prompter copies the parameters to user domain storage before calling the program.

4.1.3.5.3. CA Support for Deletion of User Profile

CA provides a macro for use when a user profile is deleted. This macro destroys all CA spaces on the user's CA queue and then destroys the CA queue.

4.1.3.6. Request Processors

Request Processors form part of the OS/400 user interface. From a user viewpoint, there are two user interfaces to OS/400: menus and CL commands. Menus can provide prompts for entry of CL commands, which are passed to the CA by Request Processors.

In general terms, a Request-Processing program receives the type of message called a request message, which requests a function from the Request-Processing program. The CL command Receive Message and the User Domain System State (UDSS) API Receive Program Message allow the program to receive the request message because both contain the appropriate MI instruction. The program can also include the Send Program Message CL command or the Send Program Message API, but including these does not make the program a request processor.

The program becomes a Request Processor when it receives a request message. When it becomes a Request Processor, it has two special features that other programs do not. First, when programs called by the Request Processor terminate abnormally, control is transferred back to the Request Processor by an escape message, and the logic of the Request Processor determines how to handle the escape message. Second, a user at the terminal interacting with the Request Processor program can use the CL command End Request to terminate processing back to the Request Processor. This CL command End Request is used, for example, when the user at the workstation presses the System Request key. The effect of the request message sent by the CL command End Request is to cancel everything on the call stack back to the Request Processor itself. Again, an escape message is sent to the Request Processor. Request Processors monitor for the escape message and perform necessary cleanup on open files and static storage.

The program remains a Request Processor until one of the following happens:

- The program ends (either normally or abnormally)
- A CL command Remove Message or the API Remove Program Message is run to remove all the request messages from the Request Processor's program message queue
- A Request Processor receives a request message but the programs's logic determines that the program should not become a Request Processor. In this case, the request message is removed from the Request Processor's program message queue as above.

A Request Processor allows the entry of commands and passes the commands to the CA.

QCMD runs in system state and is the default initial program run after the login process has completed.[14] When called, it checks its message queue for any unprocessed request messages. Commands can be placed on the message queue in several ways, as described in section 4.1.3.7 "Message Handling" on page 4-55. When there are no more unprocessed request messages on the message queue, then QCMD displays the command entry screen for interactive jobs or terminate for batch jobs.

[14] If the routing program defined in the subsystem description object routing entry is QCMD, then an initial program specified in the User Profile external object is run. If a program other than QCMD is defined in the routing entry of the subsystem description, then the program logic of this other program is followed. QCMD displays the initial menu specified in the User Profile external object. In the evaluated configuration, QCMD must be the routing entry of the subsystem description. If a program other than QCMD is used as the routing program, that is, if the system is not in the evaluated configuration, then the initial menu is not displayed after the login process has completed. See section 4.1.3.3.2 "Subsystem Monitor Jobs" on page 4-27 for more information about routing entries.

When a command is entered on the command entry panel, QCMD places it on its own message queue as a request message. This is for several reasons. First, when it receives the request message, QCMD becomes a Request Processor. Second, the request message is marked as old, and when QCMD is invoked lower in the invocation stack, the request message is not reprocessed, preventing looping.[15] After receiving the request message and marking it as old, QCMD calls the CA using the MI instruction Call External (CALLX). The processing of the command by the CA and the CPP that it calls either exit normally or not normally. When it exits normally, QCMD gets control at the instruction following the call to the CA. When the command exit is not normal, then QCMD gets control because of an escape message. In this way, QCMD passes all commands that are keyed in to this screen to the Command Analyzer. All other Request Processors supporting command entry work in the same way.

4.1.3.6.1. The User Interface Manager

The User Interface Manager (UIM) is part of the User Interface Manager (UI) component. The UIM runs System Domain System State (SDSS), except for the APIs that it includes, which run UDSS. The UIM Help Component (UH Component) provides users of the UIM with a consistent method for defining and displaying help text. The UI component provides interfaces to the *PNLGRP, *MENU, and *SCHIDX external objects for their creation, modification, and deletion. These external objects have been introduced in section 4.1.1.1 “External Objects” on page 4-2. These objects are always created by UIM in system domain. More information about each of these external objects is given in the paragraphs that follow.[16] The primary purpose of the UIM is to provide dialog manager functions for IBM internally generated screen displays, although it is available for user-written programs. IBM uses the UIM to create menus and to support menu processing. The UIM also allows use of Application Programming Interface (APIs) and exit programs in panel group objects.

A panel is a visual representation of data on the screen. A panel group object is an external object of type *PNLGRP that contains a collection of display formats, print formats, and help information. Panel types include menus, information displays, and displays that allow the entry of data. A menu object is an external object of type *MENU that contains the definition of a panel containing one or more options. Both *PNLGRP and *MENU objects are created using the UIM tag-based definition language, accessed through CL commands belonging to the UI component. An information search index object (an external object of type *SCHIDX) is created by the UIM (again, through use of a CL command belonging to the UI component) to contain search objects extracted from online help information.

All external objects (that is, of types *PNLGRP, *MENU, and *SCHIDX) created by UIM are created in the system domain and have no interfaces except through the UIM. Authorities work as described in section 6.2.1 “Authorities” on page 6-4. Use of UIM functions is controlled by authorities and special authorities that are checked when the CL commands provided by the UI component are used. For example, to create an external object of type *PNLGRP, *MENU, or *SCHIDX, users must have *USE authority to both the CL command and *READ and *ADD to the library into which the external object will be placed. In addition, to manipulate the external object of type *PNLGRP, *MENU, or *SCHIDX after creation, the user must also have *OBJEXIST to the object and *EXECUTE to its library.

The UIM can be compared in functionality to window managers such as MOTIF, although the design and implementation are completely different, because the UIM formats displays of menus, lists, data and help text displays, performs scrolling, and stacks and unstacks windows (or menus).

[15] Old (that is, processed) request messages are left on the message queue so that they appear in the job log. They can also be retrieved and displayed on a command line display when a user presses the F9=Retrieve function key.

[16] The UIM also supports programs that use Data Description Specifications (DDS) to present panels. DDS is used to build screen displays, but does not provide interfaces to *PNLGRP, *MENU, or *SCHIDX objects. Instead, the functions supported by UIM’s use of these objects must be provided by the program using the DDS panel. DDS requires input of specific screen locations for display data placement. Both DDS and UIM must format the data stream for FM to display. DDS is introduced in section 4.2.1.2.2 “OS/400 Components” on page 4-75.

The UIM provides dialog commands for screen management, contextual online help, pop-up windows, menu bars, a command line for entering CL commands, and a way to tailor the contents of a panel for different users or environments. It also provides a definition language for describing data and panels, a compiler to create panel group objects and menu objects using the definition language, and a set of APIs to display and print panels. The UIM handles function key presses defined to it by using dialog commands, some of which are discussed below. Others, such as the dialog commands for rollup, rolldown, and similar display manipulation, are not discussed further. Dialog commands differ from CL commands in that they cannot be entered on a command line and are valid only within the scope of the UIM. Dialog commands are accessed through the CL commands for *PNLGRP, *MENU, or *SCHIDX external object creation and manipulation described above, or through the set of APIs provided by the UI component.

4.1.3.6.2. UIM Structure

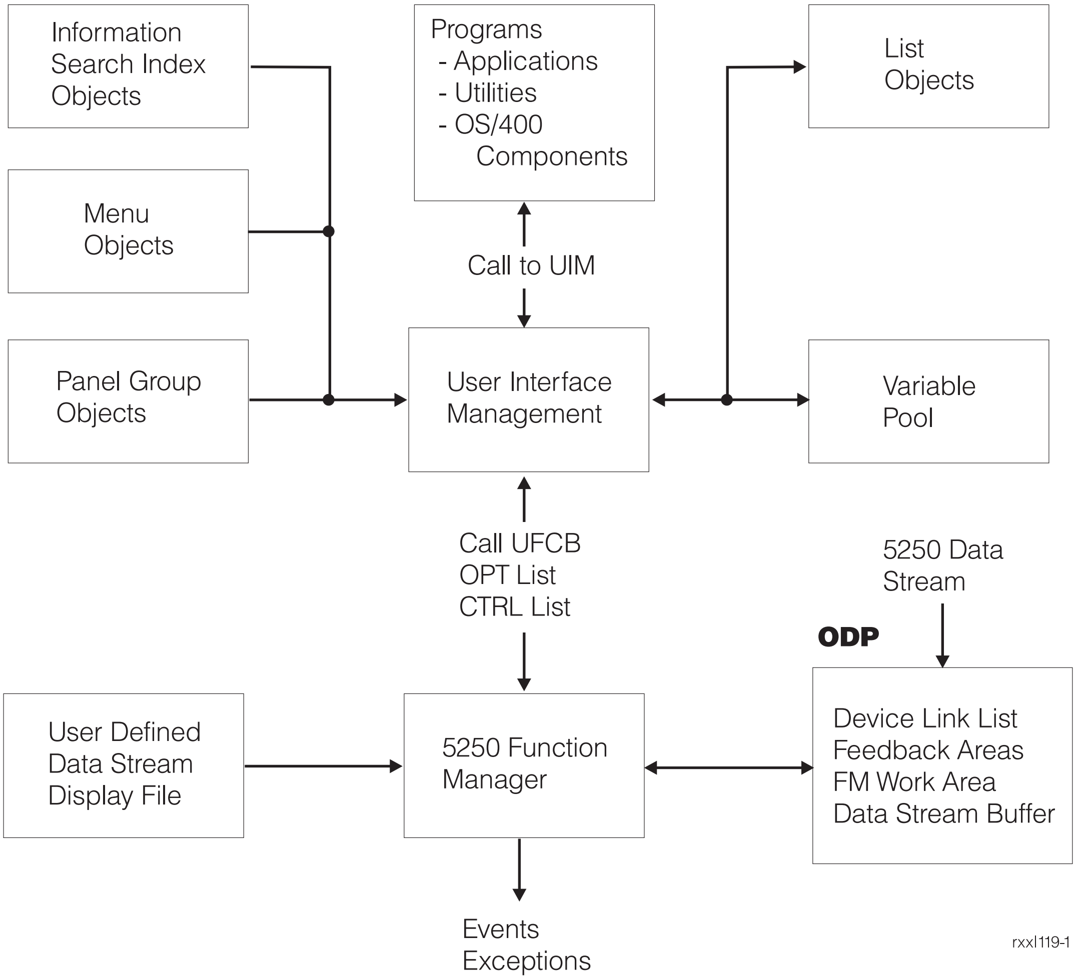


Figure 4-11. UIM Structure

Figure 4-11. “UIM Structure” on page 4-53 shows the structure of the UIM. The following paragraphs discuss this figure, starting from the upper right-hand corner.

List objects are collections of dialog variables, whose values come from the variable pool. The variable pool is the set of all dialog variables for the application. A dialog variable is a UIM element that contains a value.

This value can be referred to and updated by programs that use the provided APIs and by the UIM when it displays panels to users or performs similar functions. Some dialog variables, known as Z-variables, can be retrieved but not modified.

The *SCHIDX, *MENU, and *PNLGRP external objects provide input to the UIM for display screens. The UIM is called from a user state or system state program when screen display manipulation is needed. The UIM opens the workstation file, obtains a UFCB and ODP, and then calls the FM to complete I/O to the output device. The UIM formats a data stream and places it in the ODP buffer. This data stream includes screen images and data to be displayed. The UFCB and the ODP are passed to the FM.[17] User input from the workstation display screen uses the same path, but in the reverse direction.

Dialog commands are special functions, recognized only by the UIM, that equate actions entered by a user with screen management functions. Dialog commands differ from CL commands in that they cannot be entered on a command line nor are they valid outside the scope of the UIM. The dialog commands are included with the *PNLGRP object definition. Among the dialog commands for screen management are the following:

- CALL, which calls a program outside of the UIM
- CMD, which submits an AS/400 CL command for processing
- RETURN, which returns control (and a return value) to the program calling UIM for processing

The CALL dialog command is provided so that an application can link programs and panels without a command interface. The CALL dialog command makes program calls to programs that handle function key presses, menu item selection, action list options, or pull-down field choices. The UIM uses the current value of the CALL dialog variable to identify the program to call. In addition, exit programs allow applications to perform functions (such as editing of input fields) for which the UIM does not provide generic support. Generalized or special exit programs are named on the USREXIT attribute of the PANEL tag.[18] The usual authority checking takes place for all program calls made by the CALL dialog command.

A job can contain many UIM applications. Each Work With CL command, each display screen, and each command line entry prompt can be a separate UIM application within the same job. Multiple UIM applications can be open at the same time. When a UIM application is opened, a *PNLGRP object for that UIM application is opened. In addition, UIM assigns a unique eight-character identifier to the application. This identifier is returned to the calling program and used to tie the calling program to the UIM application for further processing.

The application is opened by an Open Application API.[19] When an application is opened, the program state asserted on the call to the UIM component is saved as the application state. The application state is kept as a variable within the (AMCB) for that application.[20] Each open UIM application has its own set of UIM data structures kept in its AMCB.

The application state is used by UIM as follows. When the UIM calls an exit program or uses the CALL dialog command, the possibility that the called program should run in user state exists. The UIM uses the application state to set the state of exit programs that can be called for the open application. (For more discussion of exit programs, see section 4.1.3.14 “Exit Points, Exit Programs, and User Exits” on page 4-67.) The UIM specifies dropstate on the MI instruction Call External (CALLX) when the UIM application was opened by

[17] Figure 4-11. “UIM Structure” on page 4-53 labels the data stream and FM as “5250”. “5250” is a generic name used for all 525x non-programmable workstations. The UIM is designed to support 525x non-programmable workstations only.

[18] Actually, the USREXIT attribute specified which dialog command to process. A parameter of the CALL dialog command is the name of the dialog variable. The contents of this dialog variable actually indicate the exit program to be called.

[19] There are two Open Application APIs, one for display applications and one for print applications.

[20] The AMCB is chained off the Job Main Control Block, which is chained off the Work Control Block for the job. See section 4.1.3.3.1 “Job Structure” on page 4-24 for more on this.

user state code. When the application was opened by system state code, the UIM does not specify dropstate. (When dropstate is specified, only programs that user domain can be successfully called.) For more discussion of program state, see section 4.1.2.2 “Programmatic Interfaces” on page 4-15.

The UIM does not suppress adoption of authority. This is because the program that called the UIM (see Figure 4-11. “UIM Structure” on page 4-53) may need to pass on adopted authority to the exit program called by the UIM. The UIM merely lets this adopted authority pass through. The UIM never adopts authority itself.

4.1.3.6.3. The UIM as a Request Processor

When a program belonging to the UIM component becomes a Request Processor, the application running is isolated from commands entered on the command line. Instead, these commands are handled as described above in section 4.1.3.6 “Request Processors” on page 4-51. In the following three cases, the UIM automatically becomes a Request Processor:

- For any panel with a command line. The UIM is required to be a Request Processor when submitting a CL command from a command line.
- For any full-screen panel with an action list. Since most action list panels have a command line, the UIM becomes a Request Processor for consistency.
- For any full-screen panel with a menu area, since most menu areas have a command line. This is again for consistency.

The UIM module QUICMD is called by other UIM modules to invoke the CA or a user program. It becomes a Request Processor when it is called to run a command that originated on a UIM command entry line. It runs, like the rest of the UIM (except for the APIs), in SDSS. There are no external APIs to call QUICMD. It is called only by the UIM.[21]

4.1.3.7. Message Handling

At the OS/400 level, the Message Handling (MH) component handles both interprocess communication (IPC) and also exception handling. (See section 3.2.3.6 “SLIC IPC Management” on page 3-72 and section 3.2.3.8 “SLIC Exception Management” on page 3-96 for how the SLIC layer handles these two functions.) In addition, the MH component provides the QCMD program for CL command entry; QCMD and other request processor programs are described in section 4.1.3.6 “Request Processors” on page 4-51. The MH/MHQ component provides interfaces to the external objects: *MSGF, *MSGQ, and *DTAQ.

The paragraphs below discuss messages in general. Next, the support for IPC is described, including handling of message files, message queues, and data queues. This is followed by a description of MH support for exception handling.

4.1.3.7.1. Messages

Messages can be used to allow processes to exchange information or to synchronize execution. In addition to sending messages, programs can receive messages, put the received message into a program variable, remove messages from a message queue, and monitor for messages.

[21] In addition to QUICMD, two other UIM programs, QUIMGFLW (Manage Flow) and QUIMNDRV (Menu Driver), also become Request Processors to support command entry.

Messages can be defined as one of the following types:

- Informational. A message that conveys information about the condition of a function.
- Inquiry. A message that conveys information but also asks for a reply. The system reply list can be used to reply to an inquiry message automatically.
- Notify. A message that describes a condition for which a program requires corrective action or a reply from its calling program. A program can monitor for the arrival of notify messages from the program it calls.
- Reply. A message that is a response to an inquiry or notify message.
- Sender's copy. A copy of an inquiry or notify message that is kept by the sender to obtain the reply.
- Request. A message that requests a function from the receiving program.[22]
- Completion. A message that conveys the completion status of work.
- Diagnostic. A message about errors in the processing of a system function, in an application program, or in input data.
- Status. A message that describes the status of the work done by a program. A program can monitor for the arrival of status messages from the program it calls.
- Escape. A message that describes a condition for which a program must end abnormally. A program can monitor for the arrival of escape messages from a program that it calls.

Messages of types status, notify, or escape are exception messages and are handled by the exception management routines described in section 4.1.3.7.3 "Exception Handling" on page 4-58. The messages of other types are used for interprocess communication and are described immediately below.

4.1.3.7.2. Interprocess Communication

A message can be predefined or immediate. Interactive users can send only immediate messages and replies, but programs can send immediate messages as well as predefined messages with user-defined data. Predefined messages are stored in a message file. An immediate message is created by the sender at the time it is sent, and is not stored in a message file.

4.1.3.7.2.1. Message Files

A predefined message is stored in a message file. A message file is an external object of type *MSGF. Each predefined message in a message file is uniquely identified by a 7-character code and is defined by a message description. The message description includes attributes such as the text, help text, severity level, and valid and default reply values. The CL command Add Message Description (ADDMSGD) adds a message to a message file. Predefined messages are used by both the TCB and user-written applications for error messages, log entries, or other repeated situations.

4.1.3.7.2.2. Message Queues

All messages that are sent or received in the system are transmitted through a message queue. A message queue is an external object of type *MSGQ. The user or program associated with the message queue receives the message from the queue. A reply to the message can be sent back to the message queue of the user or program requesting the reply.

[22] For example, a CL command can be a request message. A request processor handles a request message; see section 4.1.3.6 "Request Processors" on page 4-51 for further information about request processors.

The types of message queues, and the uses of each type, are as follows:

- Work station message queues are used for sending and receiving messages between work station users and between work station users and the system operator. The name of the queue is the same as the name of the work station. The queue is created by the system when the work station is described to the system.
- User profile message queues can be used for communication between users. User profile message queues are automatically created in library QUSRSYS when the user profile is created.
- Job message queues are used for receiving requests to be processed and for sending messages that result from processing the requests. An example is a CL command submitted by a user. The job messages are sent to the requester of the job. Job message queues exist for each job and exist only for the life of the job. Job message queues consist of an external message queue and a set of program message queues.
- The system operator message queue (QSYSOPR) is used for receiving and replying to messages from the system, display station users, and application programs.
- The history log message queue (QHST) is used for logging or saving information to the history log from any job in the system. Any message sent to QSYSOPR is automatically sent to QHST. In addition, any job can also send a message to QHST.
- User-created message queues are used for sending messages to system users and between application programs.

Since message queues are external objects of type *MSGQ, and message files are external objects of type *MSGF, both of these types of containers for messages are protected by DAC. Messages are auditable. Other restrictions are placed on the sending and receiving of messages by the program state and object domain attributes; a summary of these restrictions can be found in section 7.1.2 “Resource Isolation and Protection” on page 7-3. System-created message queues are in system domain, while user-created message queues are in temporary storage in user domain.

4.1.3.7.2.3. Data Queues

Data Queues are external objects of type *DTAQ, and are created within objects of type *LIB. Data queues are the fastest means of asynchronous communication between two jobs. Using a data queue requires less overhead than using database files, message queues, or data areas to send and receive data between jobs. Data queues are created by use of the CL command Create Data Queue (CRTDTAQ).

APIs are called to send to, receive from, and clear a data queue from a high-level language program. A job receiving data from a data queue can wait until an entry is received by use of a time-out value. Any number of jobs can send to the same data queue, and more than one job can receive data from the same data queue. For example, if several printers are available to print orders, several interactive jobs could send requests to a single data queue. A separate job for each printer could receive from the data queue. In addition, a data queue can be used to communicate within a job instead of between jobs.

When a entry is received from a data queue, the entry is removed from the data queue but the auxiliary storage is not freed. The same auxiliary storage is used again when a new entry is sent to the data queue.

Restrictions on data queues are similar to the restrictions on message queues.

4.1.3.7.2.4. Semaphores

A semaphore is a synchronization mechanism similar to a mutex or a machine interface (MI) lock. The purpose of the semaphore service is to provide OS/400 UNIX-type processes with a way to synchronize actions with each other through a set of standardized APIs. These APIs are based on the definitions in the X/Open single UNIX specification - formerly Spec 1170. Semaphores can be used to coordinate the sharing of resources or to notify other processes of the availability of resources.

Semaphores are identified by a unique positive integer called a semaphore identifier. Associated with each identifier is a data structure that contains information (integer values) for a semaphore. These values relate to the process that owns and the process that created the semaphore. This structure is a piece of storage allocated in system domain. User applications can access the data structure only via the OS/400 semaphore functions (APIs). The data structure is created and initialized by OS/400 on behalf of the process that issues the create semaphore API. When a semaphore identifier is removed from the system, the corresponding data structure is destroyed.

A process gets a semaphore identifier by calling one of the OS/400 semaphore APIs. Depending on the parameters passed in, either a new semaphore is created or an existing semaphore is accessed. Once a process gets an identifier, the information in the corresponding data structure is checked before allowing that process to perform any subsequent IPC operations. There are three semaphore functions (APIs). They are all entry points into one service program called QP0ZSCPA. This service program is user domain/system state and does not adopt any authority. IP audit records are generated for all semaphore creations, deletions, and access attempts.

4.1.3.7.3. Exception Handling

An exception is communicated from the VLIC layer across the MI through an MI instruction Retrieve Exception Data. An exception description is used to determine what data should be retrieved.[23] User state programs and OS/400 itself may raise exceptions above the MI when errors are detected above the MI, either by use of the MH-provided CL commands or the MH-provided APIs.

Resume points can be established for use when a program successfully handles a raised exception. If the raised exception is not handled, default action is taken by the OS/400 exception handling component.

For OMI programs, an unhandled exception generates a special escape message known as a function check message. The escape message is sent to the call message queue of the call stack entry that raised the original exception. If this process cannot handle the escape message, the escape message is sent up the stack to previous call stack entries until the escape message is handled. If the escape message is never handled, the job is terminated.

For NMI programs, the unhandled exception message is sent to the previous call stack entry message queue until the exception is handled or a control boundary is reached. A control boundary is an ILE call stack entry in a different activation group, or a call stack entry that is an OMI program. When the control boundary is reached, the exception message is converted to an escape message known as a function check message, and the resume point is used to define where processing of the escape message should continue. The resume point is the exception handler.

If the call stack entry receiving the exception message is running in user state, a check is made to verify that the call stack entry issuing the exception message is either a request processor, or that the call stack entry issuing the exception is a user exit. If the call stack entry is a request processor, then it is notifying the user state program of the completion of work done on the user state program’s behalf. See section 4.1.3.14 “Exit Points, Exit Programs, and User Exits” on page 4-67 for more about user exits.

[23] The exception description is defined in the ODT.Exception descriptions are valid only when the program defining them is invoked.

4.1.3.8. OS/400 Event Management

Events are handled by OS/400; no system interfaces allow user state programs to directly handle events. Events are identified as follows:

- Event class. The event can be related to a specific MI object type, or to a condition, or to a machine function.
- Event type. The type further defines the class. For example, one type of the condition class would be the error log filling up, and one type of machine function would be machine power.
- Event subtype. The subtype further defines the type. For example, a subtype of the machine power type would be a power controller failure.

Event monitoring can be done at the class level, at the class and type level, or at the class, type, and subtype level, and can be qualified. For example, the SLIC QM component might monitor for all events of class error condition and the type specifying any queue object filling up. A SLIC component using a specific MI queue object might monitor for the same class and type and further specify the name of the specific MI queue object.

Classes, types, and subtypes are all identified by hexadecimal values. (Type 00 is reserved to indicate generic monitoring of all events in that class, and subtype 00 is reserved to indicate all event subtypes in that type.) Event monitoring for specific objects uses the system pointer to the object as a qualifier. Events can also use qualifiers that specify time; for example, a qualifier can specify a time of day or an elapsed time to wait. Events for processes can use the process identification as the qualifier.

The event monitor is established by the MI instruction Monitor Event and terminated by the MI instruction Cancel Event Monitor. (Event monitors are also terminated by process termination.) The event monitor can establish priorities for events and can be enabled or disabled. In addition, the event monitor can identify an event handler to be given control when the event is signalled. Data related to the event is available to both the event monitor and the event handler; examples of event-related data include the time of the event signal, the origin of the signal, the number of signals pending, and event-specific data.

The event signalling portion of the event management component defines the occurrence of an event, and then communicates that occurrence to all qualifying event monitors. Events can be signalled directly to a process or can be signalled using the MI instruction Signal Event.

Once events are signalled, the event handling portion of the event management component calls the identified event handler. The event handler can cause the invocation of a designated process, or can allow testing for the occurrence of an event by the MI instructions Test Event and Wait on Event.

4.1.3.9. OS/400 I/O Management

Figure 3-19. “High-level View of I/O” on page 3-80 in section 3.2.3.7 “SLIC I/O Management” on page 3-79 introduces the conceptual view of I/O processing in AS/400. Figure 3-22. “High Level View of Physical I/O Layer” on page 3-87 illustrates the components needed to support I/O for workstations and printers, including the Function Manager (FM). Figure 3-24. “Detailed View of I/O for Tape, Optical and Diskette Devices” on page 3-95 does the same for tape and diskette devices, while Figure 3-25. “Detailed View of I/O for DASD” on page 3-95 provides a view of DASD I/O. Figure 4-12. “I/O Operation” on page 4-60 presents another view of the entire I/O process; this figure concentrates on the data structures used at each layer instead of the components. The sections that follow discuss Function Management and the other above-MI components, the data structures used for I/O across the MI, and finally give an example of I/O as it appears from OS/400.

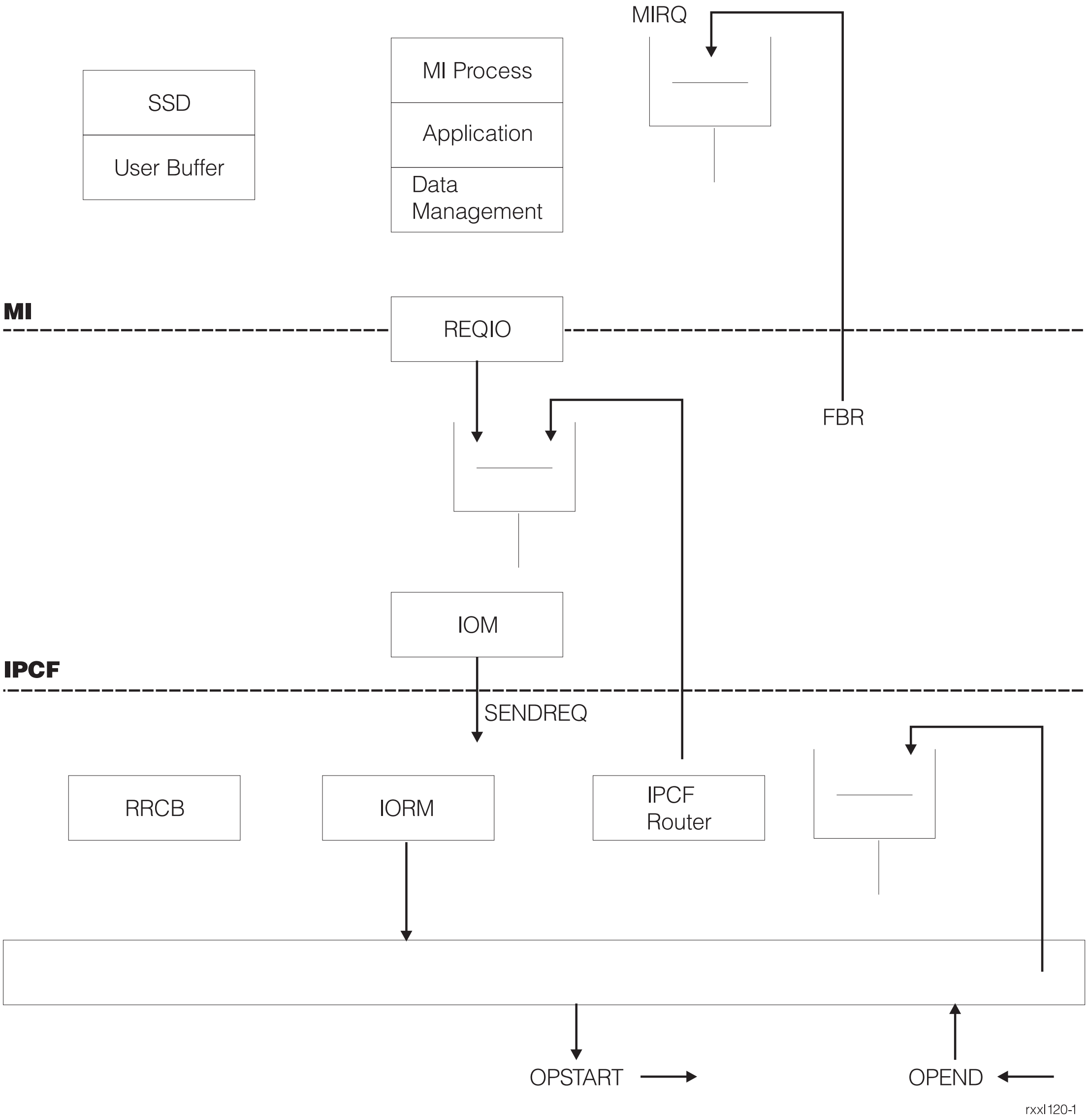


Figure 4-12. I/O Operation

4.1.3.9.1. Function Management

The various FM components provide APIs and CL commands that allow end users and other programs at the OS/400 level to request that I/O be performed. One basic function of an FM is to format an MI instruction Request I/O (REQIO) and pass it across the MI to the Source/Sink. The MI instruction REQIO consists of an opcode and a pointer to the Source/Sink Request (SSR) data structure. This SSR contains the information required for each I/O request and is described in greater detail below. The other basic function of an FM is to pass the requested data (or error notification), or the notification of successful or unsuccessful completion of data storage, back to the requestor upon completion of the I/O.

The three figures in section 3.2.3.7 “SLIC I/O Management” on page 3-79 showing the components at all layers that support I/O show that the three types of I/O require different support from FMs. These differences are summarized in the following paragraphs.

Figure 3-24. “Detailed View of I/O for Tape, Optical and Diskette Devices” on page 3-95 shows that the FM for diskette I/O, DK, and the FM for tape I/O, TA, format the MI instruction REQIO and pass it across the MI directly. Both FMs provide macros for open, get, put, close, and similar operations. In addition, these FMs provide CL commands to erase, label, and otherwise manipulate these devices.

Figure 3-24. “Detailed View of I/O for Tape, Optical and Diskette Devices” on page 3-95 shows that the FM for workstation and printer I/O passes the MI instruction REQIO that it has built to the SNA-T3 layer, which adds parameters for routing control to the SSR. The SNA-T3 layer implements Data Flow Control for the SNA. This layer is concerned with the routing and movement of data units between origins and destinations. This layer does not examine, use, or change the contents of the data units passed from or to the FM layer. This separation, where the routing of a data unit is independent of the contents of the data unit, means that a variety of physical connections can be supported. The SNA-T3 layer passes the MI instruction REQIO across the MI to the Source/Sink SLIC component.

Figure 3-25. “Detailed View of I/O for DASD” on page 3-95 shows no FM; instead, the REQIO instructions are formatted by the Data Management component as part of file manipulation for SM or by the Save/Restore component for the Dump Space IOM. See section 4.2 “Database Management System” on page 4-68 for two examples of database file opens. See section 4.3.2 “Backup and Recovery” on page 4-90 for more on Save/Restore.

The DM component does file manipulation of all files in AS/400 and therefore supports almost all I/O. An example where DM would NOT be involved would be in the formatting of a diskette; in this case, an end user could use a CL command provided by the DK FM to format the diskette. If, however, the end user copied a file to the diskette, DM would be involved in opening the file on DASD, copying data from that file to the SSD, creating the file on the diskette (if needed), and opening the diskette file. DK would be involved in creating the SSD, formatting the data in the SSD, issuing the REQIO, and waiting for the completion of the I/O.

4.1.3.9.2. I/O Data Structures

Before the MI instruction REQIO can be issued, several data structures must be set up or located. These include the following:

- A device description external object, *DEVD, which is the OS/400 external object built upon the LUD MI object. The *DEVD external object is introduced in section 4.1.1.1 “External Objects” on page 4-2.
- A Source Sink Request (SSR). An SSR is a set of pointers and other data needed as part of the I/O request. It includes a system pointer to an MI response queue (MIRQ), a system pointer to the device description (*DEVD), and a space pointer to a source-sink data buffer. It also contains data such as a timestamp, a request priority, a request ID, and the type of I/O requested.
- A Source Sink Data (SSD) buffer. The SSD is a user buffer that contains either the data to be output or the data that is retrieved, depending on the I/O operation.
- A Feedback Record (FBR). The FBR contains a pointer to the requesting REQIO’s SSR. It also contains a request ID that matches the request ID in the SSR.
- An MI Response Queue (MIRQ). The MIRQ is used for notification that the I/O operation has completed. When I/O has completed, the IOM posts an FBR to the MIRQ. The FM can dequeue the FBR, and, if data was requested, obtain the pointer to the SSD (which now contains the retrieved data) in the SSR and pass this pointer to the requestor of the I/O.

Two other data structures are important in I/O processing. The DM component creates the Open Data Path (ODP) for a file during its open processing. This ODP is needed by the FM for all I/O to or from a file. It contains information about the presentation of the file and pointers to work areas such as that for the FM. For example, for a printer file, the ODP would contain values such as page size and font for printing the file.

Another structure that is passed to the DM, which passes it to the FM during I/O processing, is the User File Control Block (UFCB). The UFCB is created at compile time for every file used in a program. It contains information such as a pointer to the ODP, a pointer to the SSD (during and after I/O operations), and a pointer to work areas. It also contains the file name and library name for the file, and is the source of the file-specific information (such as the page size) that is copied to the ODP by DM on a file open. When an ODP is created in user domain for a user domain file, it must be hardware-storage protected.

4.1.3.9.3. I/O Example

The general sequence used to perform I/O, from the OS/400 perspective, is as follows:

- Create an MIRQ.
- Lock the *DEVD for the requested device.
- Format and send across the MI an MI instruction Modify Logical Unit Description to the LUD represented by the *DEVD, to establish an active session with the device.
- Issue an MI instruction REQIO. This instruction points to an SSR, which specifies the LUD to be used, the MIRQ on which to post the FBR, the type of I/O operation to perform, and the SSD that contains data to be stored or into which the data will be returned.[24]
- Issue an MI instruction Dequeue against the MIRQ to wait for the FBR. Once the FBR is received, determine the disposition of the I/O requested.
- Continue issuing REQIOs paired with Dequeues until the I/O is completed.
- Issue an MI instruction Modify Logical Unit Description to deactivate the session.
- Unlock the *DEVD to make the device available to other users.
- Destroy the MIRQ.

This general sequence is capable of variation, depending on the requirements of the specific I/O requested. The example given below is of I/O to a printer file. This file type is introduced in section 4.1.1.1 “External Objects” on page 4-2. The sequence is as follows:

- DM receives an open request for the printer file and is passed the printer file’s UFCB.
- DM creates an ODP, using information from the file’s UFCB.
- DM passes the ODP to the Presentation Layer component, the FM for workstations and printers.
- The FM expands the ODP to include the FM workarea and other information specifically for printing, such as the print work area.
- The FM creates the SSR, the SSD, and a request block for T3.
- With everything set up for I/O, the FM returns to DM, and DM returns to the program that requested the file open.

[24] Examples of the type of operation are normal device I/O, Load/Dump, service operations on devices such as Load IOP, or MSCP routing control.

- The program that requested the file open now requests a put of data to the printer file, and calls FM directly. It passes the pointer to the ODP created at file open, which it obtains from the file's UFCB.
- The FM takes the data from the requestor and places it in the SSD. It then passes the SSR, the SSD, and the T3 request block to SNA-T3.
- SNA-T3 checks the LUD and sets the routing information.
- SNA-T3 adds the routing information to the SSR.
- SNA-T3 issues an MI instruction Dequeue against the MIRQ and waits for the FBR.
- SNA-T3 receives the FBR and returns to the FM; FM then returns the results of the I/O to the requestor.

4.1.3.10. CL Compilers

Starting in V3R1, there are two CL compilers. There is the OPM (Old Program Model) CL compiler, the "traditional" CL compiler that has been in OS/400 since its inception. Also, there is now the ILE (Integrated Language Environment) CL compiler, so that CL can be interoperable with other ILE languages. The OPM compiler is used to create an OPM CL program (*PGM object). Likewise, the ILE CL compiler is used to create either an ILE CL program (*PGM object) or an ILE CL module (*MODULE object). Both compilers create their objects from a set of CL command source within a physical file. Neither compiler enforces any of the security policies, but runs in System State and can, therefore, possibly circumvent one or more of the security policies enforced by the system. It is for that reason that this compiler is considered part of the TCB. The remainder of this section describes the functionality of both CL compilers.

During compilation, the MI instructions invoked by the CL compiler perform DAC checks on the *CMD objects that are accessed. The user must have *USE authority (plus any required special authority) to each *CMD object included in his/her CL program. During execution of the CL program, DAC checks are performed on both the *CMD objects and the associated CPPs (*PGM objects). The user must have *USE authority (plus any required special authority) to each object.

CL commands can either be entered directly from the command line, or they can be combined into a CL program or CL module. To create a CL program or module, a user uses an editor to create a CL physical file (*FILE object) containing CL commands and CL compiler directives such as DCL, GOTO, and IF. The user then either invokes the CL command CRTCLPGM to create an OPM CL program, the CL command CRTBNDCL to create an ILE CL program, or the CL command CRTCLMOD to create an ILE CL module, from the CL source physical file. The CL command CRTCLPGM invokes the OPM CL compiler, while both the CL command CRTBNDCL and the CL command CRTCLMOD invoke the ILE CL compiler.

The CL compiler reads each CL command from the CL source physical file and invokes the Command Analyzer (CA) (see section 4.1.3.5 "Command Analyzer" on page 4-47) to perform command processing on the command receiving back a positional list. The CL compiler uses the positional list to generate a token list, which is stored with the program. For each compiler directive, the OPM CL compiler generates Intermediate Representation of a Program (IRP), and the ILE CL compiler generates WCODE.[25] Since the CL program will run in User State, the CL compilers generate calls to the UDSS CA program using the token list as the input. (When the CL program executes, CA is called to execute each CL command.) Again, the output of the first step for the OPM CL compiler is IRP; for the ILE CL compiler, it is WCODE.

The OPM CL compiler then calls the Program Resolution Monitor (PRM), which takes the IRP as input. PRM validity checks the IRP and, if there are no errors, creates a program template, which it passes to the

[25] WCODE is a common intermediate language that all the ILE compilers generate. WCODE is an input to the Common User Back End (CUBE-3) program which generates MI.

MI instruction (CRTPG). As described in section 3.2.3.10.2 “OMI Program Translation” on page 3-102, the MI instruction CRTPG invokes the translator. The output of the MI instruction CRTPG is a UDUS *PGM object. PRM returns to the OPM CL compiler a system pointer to this program object.

Instead of calling the PRM, the ILE CL compiler generates a WCODE stream, which is passed to CUBE-3, which executes and generates a UDUS *MODULE object. If the user is running the CL command CRTBNDCL, the compiler calls binder to create a program object from the module created by CUBE-3.

4.1.3.11. Restructured Extended Executor Language Interpreter

Restructured Extended Executor Language (REXX) is a programming language in OS/400 that has an interpreter running in system state. REXX serves as a command processing language and an applications programming language. REXX views all data as character strings. REXX does not get compiled into a program object. Instead, when a REXX program runs, the language processor directly interprets each language statement. Each of these statements are called *clauses*. These clauses are broken further into *tokens* which represent routines internal to the interpreter. These routines represent the execution path for each program. The REXX interpreter validates these tokens before execution.

The REXX interpreter is called when a REXX program is run. A REXX program can be run in the following ways:

- Issuing the CL command Start REXX Procedure (STRREXPRC)
- Issuing a CL command that has a REXX program as its CPP
- Calling the API QREXX

To remain within the evaluated configuration, source must be entered for the REXX program using the Source Entry Utility (SEU) editor.

REXX programs reside in source files. Users can either create their own source files or use the IBM-supplied REXX source files.

When users delete REXX files or remove REXX members, both the internal forms and the source members are deleted or removed. The internal form is also deleted if an existing member is changed from REXX to some other type.

Stream files are managed through Data Management; see section 4.1.3.9 “OS/400 I/O Management” on page 4-59. REXX uses standard stream files to receive input and to display output and errors. The input stream file is called *STDIN*, the output stream file is called *STDOUT*, and the error stream file is called *STDERR*. Data Management makes all DAC checks for REXX. The stream files default to the following:

- If a REXX program is being run interactively, STDIN defaults to the keyboard, and both STDOUT and STDERR default to the display.
- If a REXX program is being run in batch mode, the standard input stream defaults to the file QINLINE while the standard output and error streams default to the file QPRINT.

Stream files can be redirected as follows:

- STDIN can be redirected to a database physical file or to a named inline data file.
- STDOUT and STDERR can be redirected to a database physical file or to a printer file.

The REXX interpreter follows the calling conventions outlined in section 4.1.3.14 “Exit Points, Exit Programs, and User Exits” on page 4-67. The REXX interpreter relies on existing interfaces to perform the necessary DAC checking (see section 4.1.2.2 “Programmatic Interfaces” on page 4-15).

4.1.3.12. Debugger

OS/400 provides users with two debuggers, one for the OMI programs and one for the NMI programs. The OMI debugger cannot debug NMI programs. The NMI debugger can debug RPG, COBOL, and CL OMI programs that are created with source debug data. Source debug data is generated at compile time by using the OPTION parameter. The debugger chooses the appropriate debugger based on the program being debugged and the selected user options.

The debuggers are made up of many programs, which are user domain/system state and system domain/system state. To begin debugging, users must first put their session in debug mode. This is accomplished by using the CL command Start Debug (STRDBG). A flag in the work control block indicates whether the session is in the debug mode.

The OMI debugger performs the necessary checks, shown in Table 4-7 on page 4-65, by the code above the MI. For the NMI debugger, some checks are made above the MI and some are done below the MI, by the debug interpreter. To determine the user's authority to the program, the user's special authority, and program state, both debuggers make the checks above the MI level. To determine the domain of the space containing a variable, or if the space is write protected, the OMI debugger performs the checks in all programs that gain the address of OMI variables and write to OMI variables respectively. In the NMI debugger, these checks are made below the MI layer.

Table 4-7. The Authority required to debug a program (X = Doesn't matter).

Authority to Program	Special Authority	Program State	Debug Allowed
*USE	*SERVICE	X	Yes
*CHANGE	*NONE	User	Yes
*CHANGE	*SERVICE	X	Yes

4.1.3.13. System Values

System values contain specifications that can be used to control or change the overall operation of the system. System values are contained in a single space object in system domain storage. Work Management is responsible for creation and the internal mapping of the system value object.

As part of Work Management startup, a system pointer to the system value object is placed in the WCBT. If the system value object cannot be found or is damaged, Work Management creates a new system value object with the predefined install values.

Work Management provides internal interfaces that are used to read and change system values.

Users have two commands and one API to interface with system values. The two commands allow a user to read and change system values, and the one API allows the user to retrieve system values. The authorities needed to change each system value are listed in the Security Reference Manual [19]. For a list of security relevant system values, see Appendix G, "Security Related System Values", on page G-1. There are no APIs that allow a program to change a system value.

System values can be categorized into the following areas; also listed are some examples of what the categories contain:

- Date and Time
  - Date and time format
  - System date and time
- Editing
  - Date format
  - Decimal format
  - System editing values for currency
  - System editing values for time and date separation
- System Control
  - Attention program
  - Disconnect job interval
  - IPL-related information
  - Language
  - Sort sequence
  - UPS-related information
- Library List
  - System library list
  - User library list
- Allocation
  - Job message queue full
  - Job message queue initial size
  - Job message queue maximum initial size
  - Job message queue maximum size
  - Reclaim spool storage
  - Spooling size
- Message and Logging
  - Accounting level
  - History log size
  - Software error logging
- Storage
  - Machine pool size

- Security
  - Allow object restore option
  - Allow user domain objects
  - Audit values
  - Create authority
  - Inactive job timeout
  - Inactive message queue
  - Limit security officer
  - Password values
  - Retain server security data
  - Sign-on values
  - System security level
  - Use adopted authority

Certain system values are referenced for performance reasons. These system values are reflected in reserved segments in main storage. Other system values have no usage below the MI and thus are not reflected in the reserved segment. Changes to system values that have been determined not to change the policy of the machine are immediately reflected in the reserved system storage. Other system values such as Security Level are not reflected in the reserved area as it would change the policy of the current running machine. These types of system values are not recognized until the next IPL as they are not reflected in reserved system storage until IPL time (see section 4.3.1 “Initial Program Load (IPL) and Installation” on page 4-85).

4.1.3.14. Exit Points, Exit Programs, and User Exits

An *exit point* is a defined point in a System State program where that program calls a User State program. System State programs that contain exit points are known as *exit programs*; User State programs called by exit programs are known as *user exits*. For some exit points, IBM supplies a user exit that is called to perform default processing when the exit point is reached. For other exit points, IBM supplies no user exit, but the exit program checks whether a user exit has been identified (i.e., registered). If so, the exit program calls the user exit. User exits can be registered in one of the following ways:

- The user exit program name can be a parameter to a CL command or API (e.g., when a user-created CL program is specified when invoking the the CL command CALL, it is a user exit).
- The user exit program name can be specified in a System value.
- The user exit program name can be a parameter in a user profile (e.g., the initial program to be called when a user logs on is specified in the user’s user profile).
- The user exit program name can be a parameter in a HLL program (e.g., the signal function in C has a parameter to specify a user exit).
- The user exit program name can be specified as a parameter when adding a message description to a *MSGF object or when changing a message description.
- The user exit program name can be specified to an exit point registration facility. This registration facility allows an administrator to manage user exit programs registered to a set of exit points.

Some user exits can be registered by any user (e.g., specifying as a parameter in a HLL program). Others can be registered only by an administrator (e.g., specifying as a system value).

Exit programs drop to User State before calling their user exits so that the call is made from User State (this ensures that the called user exit is in User Domain).

Many exit programs (but not all) suppress adoption before calling the user exit to ensure that no adopted authority is passed to the exit program.

Since the TCB is exited when a user exit is called, exit programs must protect themselves when they regain control on the invocation stack. The technique they use is described below:

- Exit programs that pass parameters in writable UD storage maintain a local copy of any parameters they pass and always use their local copy. (Since user exits run in User State, parameters must be passed in User Domain storage. If these parameters are passed in writable UD storage, the user exit could modify these parameters. Therefore, parameters placed in writable UD storage must not be used by the exit program.)

Because of the complexity of the user exits, each exit point in the evaluated configuration was evaluated. That analysis is described in the following section.

4.1.3.14.1. Exit Point Analysis

The AS/400 TCB contains 154 exit points. For 66 of these, the corresponding user exit cannot be registered for one of the following reasons:

- The CL command or API used to register the user exit is public *EXCLUDE (and the administrator is warned in the TFM not to use the CL command or API, not to change the public authority from *EXCLUDE, and not to give any user private authority to the CL command or API).
- The CL command or API to register the user exit requires a special authority (and the administrator is warned in the TFM not to use the CL command or API).
- The administrator is warned in the TFM not to change the corresponding system value from its installed default.
- Message files (*MSGF) on a C2 system must have public authority set to no greater than *USE to prevent the use of corresponding user exits. The Security Features Users Guide (SFUG) contains warnings to users not to set the public authority on their message files to greater than *USE to prevent a user exit from being registered for the message file.

Therefore, in the evaluated configuration, there are 88 exit points where a user exit can be called. Of these 88 exit programs, 28 do not suppress adopt prior to calling their user exit.

4.2. Database Management System

A database is a set of objects used to store data. A database management system (DBMS) is a collection of programs that enables users to create and manipulate data in a database. The AS/400 DBMS implements the *relational data model*. The relational data model represents data in the database as a collection of *tables* (tables are known as *physical files* in the AS/400 system).

The AS/400 DBMS is fully integrated with the operating system and is part of the TCB. It runs in system state and executes blocked instructions. The functionality of DBMS is supported by the operating system. There is no single definable piece called a DBMS.

There is no separate identification and authentication mechanism for the DBMS. Furthermore, discretionary access control checks and audit are not done by the DBMS. The DBMS uses the programs provided by the security component of the system to enforce discretionary access control and audit. There is also no separate mechanism for disk management or locking. The DBMS uses the mechanisms provided by OS/400 and SLIC to perform disk management and locking. For instance, the lock management services of SLIC are used to lock *database files* and records. This chapter discusses the AS/400 DBMS architecture, DBMS interfaces, and data organization and integrity.

4.2.1. Database System Architecture

The AS/400 integrated database system can be divided into three layers, as depicted in Figure 4-13. “Integrated Database Architecture” on page 4-70. The first layer is the interface layer. There are several different and unique interfaces to the database on the AS/400 system. The OS/400 layer provides support for the database interfaces. It includes many components that are essential in executing database requests. And finally, there is the SLIC layer. The SLIC layer provides a high level machine interface to OS/400. It also provides MI objects, which are used as building blocks for OS/400 database files. In the next few sections, we discuss each of the layers in some detail.

4.2.1.1. Database Management System Interfaces

AS/400 provides end users with several interfaces to define and access data stored in database files. These interfaces differ in their capabilities to create, manipulate, and display database files. These interfaces include: Source Entry Utility editor, Structured Query Language, CL commands, Query/400, Query Manager, ILE C Language, and Application Program Interfaces (APIs).

4.2.1.1.1. Source Entry Utility Interface

The Source Entry Utility (SEU) editor is a full-screen editor that allows users to manipulate *source physical files*. Users can use the SEU to browse or print source physical files. The SEU editor does not provide any security services. However, SEU runs in system state (see section 4.3.3 “Source Entry Utility” on page 4-98).

4.2.1.1.2. Structured Query Language Interface

The query language in the AS/400 is called *Structured Query Language* or SQL. SQL is a comprehensive relational database language; it has statements to define, manipulate, and optimize a database. SQL can be used interactively or it can be embedded in a host language. SQL runs in system state.

The CL command Start SQL (STRSQL) is used to start an interactive SQL session. When SQL is embedded in an application program, the application program must be precompiled and then compiled. Precompiling of such programs is done by the SQL precompiler. The SQL precompiler scans each statement of the application program source and does the following:

- Verifies that each SQL statement is valid and free of syntax errors
- Validates the SQL statements
- Prepares each SQL statement for compilation in the host language
- Produces information about each precompiled statement (e.g., diagnostic messages)

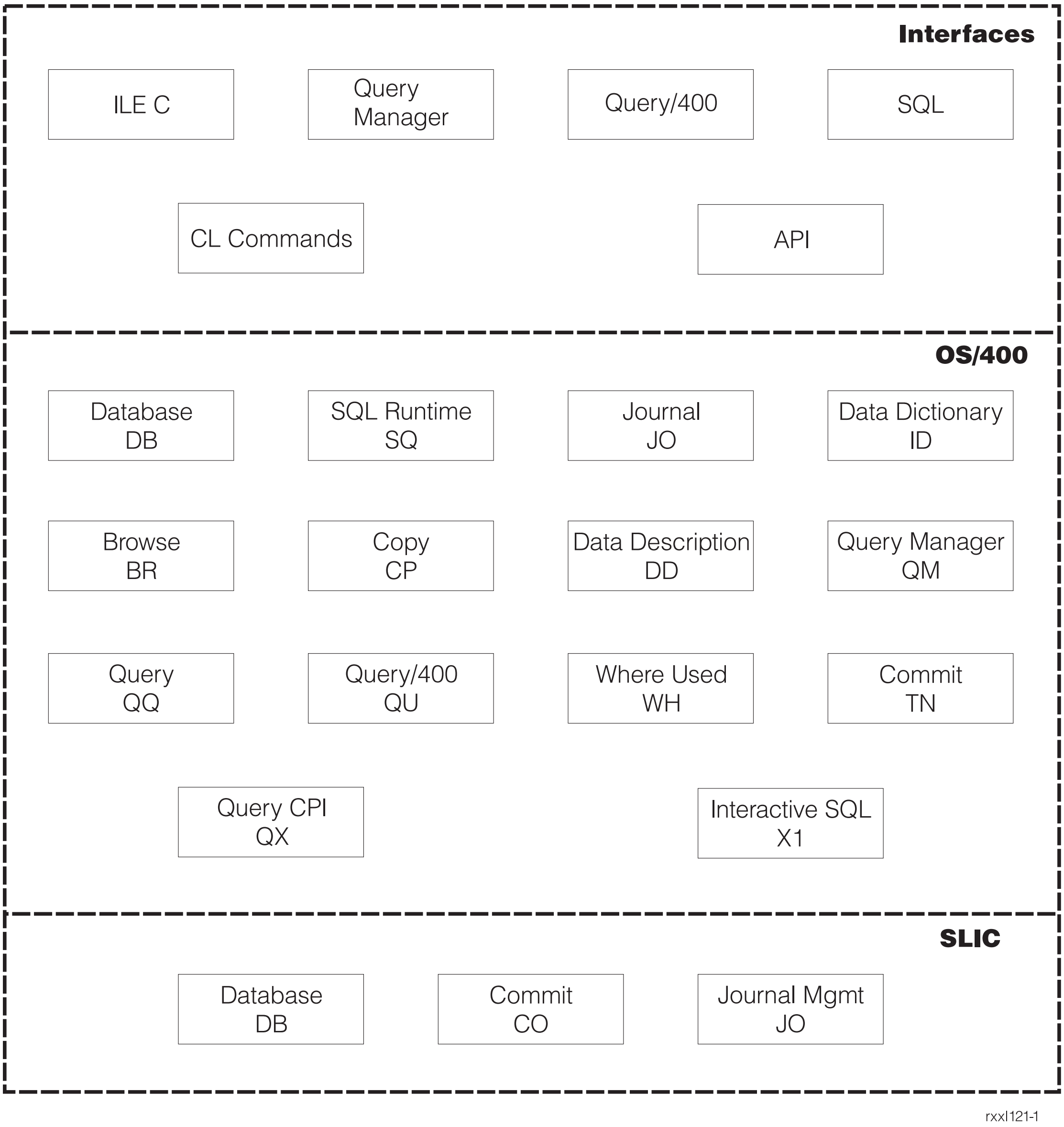


Figure 4-13. Integrated Database Architecture

SQL can be used to create *collections*, *tables*, *views*, and *indexes*. Table 4-8 on page 4-71 shows the relationship between AS/400 system terms and SQL terms.

Table 4-8. Relationship of AS/400 System Terms to SQL Terms

System Terms	SQL Terms
Library	Collection
Physical File	Table
Logical File	View
Index	Index

- **Collections:** The objects in a relational database are organized into sets called collections. A collection provides a grouping of objects in the database.

  When a table, view, or index is created, it is assigned to exactly one collection. The collection to which an object is assigned is determined by the name of the object. For example, CREATE TABLE A.B creates TABLE B in collection A. A collection is created as the following:

  - A library: A library groups related objects.
  - A data dictionary: An optional data dictionary describes object definitions (schemas).
  - A journal and journal receiver: A journal and journal receiver are created in a collection and are used to record changes to all tables subsequently created in the collection.
  - A catalog: A catalog is a set of views (based on views and tables in library QSYS2) containing information about the data in the collection. The catalog views contain information about tables, views, and indexes. Views in the catalog are like any other views. SQL statements can be issued to access data in the catalog. The catalog consists of the following views:
    - **SYSTABLES:** Describes the tables. Each row of SYSTABLES identifies a table defined in the collection. Whenever a new table is created in the collection, a new entry is automatically added to the SYSTABLES table. Each new entry contains information such as the name of the table, and the number of columns and rows.
    - **SYSVIEWS:** Describes the views. Each row in SYSVIEWS identifies a view created in the collection. SYSVIEWS is updated when a new view is defined or an existing view is modified.
    - **SYSINDEXES:** Describes indexes on tables and views. Each row contains the index name and the column numbers of the columns of the index. Creating a new index in the collection causes a new entry to be added to SYSINDEXES. Dropping an index in the collection causes that entry to be deleted from SYSINDEXES.
    - **SYSCOLUMNS:** Describes the columns in the tables. Each row contains a column name, the table name to which the column belongs, the type of the column, and the length of the column.
    - **SYSVIEWDEP:** Describes the dependencies of views on tables or other views. This table gets updated when a new view is created or an existing one is dropped.
    - **SYSKEYS:** Describes the keys. SYSKEYS contains one row for every column of an index in the collection.

  - **SYSCST:** Describes information about all constraints.
  - **SYSCSTCOL:** Describes information about the columns referenced in constraints.
  - **SYSCSTDEP:** Describes information about dependencies on tables referenced in constraints.
  - **SYSKEYCST:** Describes information about unique, primary, and foreign keys.
  - **SYSREFCST:** Describes information about referential constraints.

- **Tables:** A table is an object that stores data. Tables are made of columns and rows. At the intersection of every column and row is a specific data item called a value. A column is a set of values of the same type. The data type that describes the types of values that can appear in each column is called a domain. Rows represent a collection of related data values. Tables are created with the CREATE TABLE statement and removed with the DROP TABLE statement.
- **Views:** A view provides an alternate way of looking at the data in one or more tables. Like tables, views have rows and columns. Unlike tables, views have no storage representation. When a view is created, its definition is stored in the catalog. No data is stored. Views are created with the CREATE VIEW statement and deleted with the DROP VIEW or DROP TABLE of the base table of the view.
- **Indexes:** An index is a set of pointers to rows of a table. Indexes are used to improve performance. Indexes are created with the CREATE INDEX statement and removed with the DROP INDEX statement.

SQL users can use the GRANT or REVOKE SQL statement to grant or remove table and view authorities to or from other users. Because the GRANT and REVOKE statements assign and remove AS/400 authorities for SQL objects, each SQL authority corresponds to one or more AS/400 authorities. SQL authorities are assigned to and removed from either the table specified or, if a view is specified, to the base table on which the view is specified and on which the view is dependent. Table 4-9 on page 4-72 shows the SQL authorities granted to tables and their corresponding AS/400 authorities.

Table 4-9. Authorities GRANTed to Tables

SQL Authority	Corresponding AS/400 Authority
ALL (Only grants of those privileges you currently have)	*OBJMGT (Revoke only) *OBJREF *OBJALTER *OBJOPR *ADD *DLT *READ *UPD
ALTER	*OBJALTER
DELETE	*OBJOPR *DLT
INDEX	*OBJALTER

Table 4-9. Authorities GRANTed to Tables

SQL Authority	Corresponding AS/400 Authority
INSERT	*OBJOPR *ADD
REFERENCES	*OBJREF
SELECT	*OBJOPR *READ
UPDATE	*OBJOPR *UPD
WITH CHECK OPTION	*OBJMGT

Table 4-10 on page 4-73 shows the SQL authorities and the corresponding AS/400 authorities granted to views and to base tables.

Table 4-10. Authorities GRANTed to Views and Base Tables

SQL Authority	Corresponding AS/400 Authority Granted to a View	Corresponding AS/400 Authority Granted to Referenced
ALL (Only grants of those privileges you currently have)	*OBJMGT (Revoke only) *OBJALTER *OBJOPR *OBJREF *ADD *DLT *READ *UPD	*ADD *DLT *READ *UPD
ALTER	*OBJALTER	None
DELETE	*OBJOPR *DLT	*DLT
INSERT	*OBJOPR *ADD	*ADD
REFERENCES	*OBJREF	None
SELECT	*OBJOPR *READ	*READ
UPDATE	*OBJOPR *UPD	*UPD
WITH CHECK OPTION	*OBJMGT	None

4.2.1.1.3. CL Commands Interface

AS/400 provides users with several commands to create, destroy, and display database files. For instance, database files can be created with the CL commands Create Physical File (CRTPF) and Create Logical File (CRTLF). For more about CL commands, see section 4.1.2.1 “Command Language” on page 4-14.

4.2.1.1.4. Query/400 Interface

The Query/400 component on AS/400 provides an alternative way to interact with database files. Query/400 provides the users with a screen-oriented interface. It allows users to create, modify, save, and execute queries. In addition, users can specify the layout of the selected data. Query/400 does not provide any security services. However, it runs in system state. The CL command Work with Query (WRKQRY) is used to start a Query/400 session.

4.2.1.1.5. Query Manager Interface

Query Manager is a collection of tools that enable easier access to Structured Query Language and provides a method for defining report forms that format the output from SQL statements, and a tables facility that gives the user a ‘forms type’ of interface to database file creation and population. Essentially, Query Manager acts as a front-end to SQL. The CL command Start Query Manager (STRQM) is used to start a query manager session.

4.2.1.1.6. ILE C Language Interface

The AS/400 system also provides users with the ability to write ‘C’ language programs to manipulate database files. The AS/400 system also allows the users to embed SQL statements in a ‘C’ program.

4.2.1.1.7. Application Program Interface

The AS/400 Application Program Interfaces (APIs) are intended for use by programmers who create software applications. However, they also allow users to access data stored in database files (see section 4.1.2.2 “Programmatic Interfaces” on page 4-15).

4.2.1.2. Database Management System Components

This section describes the SLIC and OS/400 software components that support the functionality of the integrated database management system.

4.2.1.2.1. SLIC Components

- **Database (DB):** The SLIC database component provides the support for creation, destruction, and management of the MI objects that comprise database files. Namely, it supports the data space, data space index, and cursor MI objects.
- **Commit (CO):** The commit component supports commitment control, which provides a means to process a group of physical file changes as a single unit of work (i.e., transaction). Commitment control guarantees that the transaction is completed.
- **Journal Management (JO):** The journal management component provides support for forward and backward recovery. Forward recovery can be used to take an older version of a physical file and apply the changes logged on the journal to the physical file. Backward recovery can be used to remove changes logged on the journal from the physical file.

4.2.1.2.2. OS/400 Components

- **Database (DB):** The database component provides an interface between other components in the system and the SLIC database support. For example, the interface to create database files is available through SQ, CP, or other, but all these components interface with the database component to actually create the database. The ability to create, delete, and manage database files is provided through this interface.
- **SQL Runtime (SQ):** SQL runtime provides the support to execute all SQL statements.
- **Journal (JO):** The journal component provides generic support to log entries to a journal and its associated journal receiver objects. The user is given the ability to recover a database file by using the apply and remove journal changes. The commit component (TN) also uses the journal to support commit and rollback operations.
- **Dictionary (ID):** The data dictionary component provides an interactive interface for creating, deleting, and managing data dictionary objects. A data dictionary contains definitions of fields and record formats.
- **Browse (BR):** The browse component provides support to allow a user to browse the data in a file. No selection is supported.
- **Copy (CP):** The copy component provides support to copy data from one physical file to another. Several options are supported such as selection and data conversion.
- **Data Description (DD):** The data description component allows a user to create physical files by passing a *source physical file* that contains data description specifications (DDS). DDS is a language that describes the record formats of files (including physical, display, etc.).
- **Query Manager (QM):** This component provides the support, including messages and text, for the Query Manager interface.
- **Query (QQ):** This component provides the interface between other components and SLIC database support for performing queries. Thus, Query Manager, SQL, and other components all go through this interface. This component also optimizes queries in order to improve the performance of a query.
- **Query/400 (QU):** The Query/400 component provides all the runtime support for running queries that were created by the Query/400 product. This allows a user to create and save a query and then run it at a later time.
- **Where Used (WH):** The Where Used component provides commands to display or print information about database files.
- **Commit (TN):** The commit component provides transaction manager support for AS/400. Database changes can be committed or rolled back.
- **Query CPI (QX):** This component provides an API and command level interface for creating and managing queries.
- **Interactive SQL (X1):** The Interactive SQL component in OS/400 provides the prompting support for SQL statements.
- **Data Management (DM):** The Data Management component provides the functions that open and close files. See section 4.1.1.1.2 “Data Management” on page 4-9 for more information.

4.2.1.3. AS/400 Database File Structure

This section discusses the AS/400 database file structure at the OS/400 and the SLIC levels. Only OS/400 database objects are presented to users. SLIC provides the MI objects on which OS/400 database objects are built (see section 3.2.1.5 “MI Objects” on page 3-26 for more detail).

4.2.1.3.1. OS/400 Database File Structure

Users of the AS/400 system can issue transactions against database files. A database file is comprised of two parts; file description and file member as depicted in Figure 4-14. “Database File Structure” on page 4-76. All parts of database structures are system domain objects.

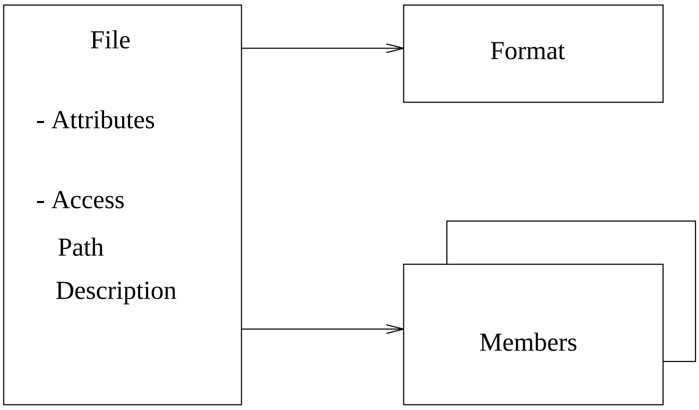


Figure 4-14. Database File Structure

- File Description: The file description contains the characteristics of the database file. It contains attributes pertaining to recovery and maintenance. Access-path description is also maintained in the file description. An access path description describes the order in which records are to be retrieved. Records in a database file can be retrieved with an arrival sequence access path or a keyed sequence access path. The arrival sequence access path is based on the order in which records arrive and are stored in the file.[26] A keyed sequence access path is based on the contents of the key fields. File description also contains record formats. The record format describes the order of the fields in each record. The records format also describes each field in detail, including length, data type, and text description.
- File Member: A database file member serves as a subset of the data in the database file. In general, database files have only one member, the one created when the file is created. However, many members can be associated with a file at any given point. A user can issue CL commands to partition the database file into several members.

[26] This is only true if the “reuse deleted records” attribute is not specified for the file.

The three types of database files at the OS/400 level are source physical files, physical files, and logical files.

- Source Physical Files: From a database perspective, a source physical file is a file that contains the specifications that describe the record format and its fields for a file. The specifications are written in the Data Description Specification language. For example, when users creates a file (e.g., a physical file), they may specify the name of the source physical file that is used to describe the layout of the physical file.
- Physical Files: A physical file is a file that contains (physically) the actual data. A physical file is comprised of several MI objects: a space, a cursor, a data space, and an optional data space index, as shown in Figure 4-15. "Physical File Member Structure" on page 4-77. (Note: The MI objects that comprise a physical file are discussed in section 3.2 "SLIC and MI" on page 3-16.)

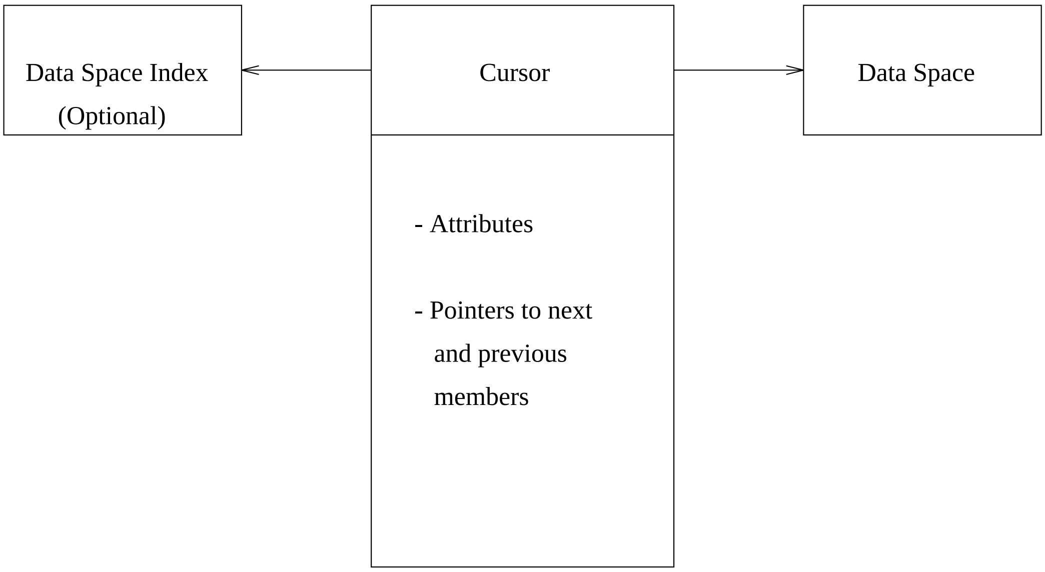


Figure 4-15. Physical File Member Structure

- Logical Files: A logical file is a single file that is derived from other physical files. A logical file does not contain data. It is considered to be a virtual file. Only the virtual file definition is stored in the file description. When a query is issued against a view, it is actually specified against the physical files that are deriving the logical file. Logical files are important for several reasons:
  - They allow record and field-level protection (see section 6.2 "Discretionary Access Control" on page 6-4).
  - They allow different ordering of records.
  - They allow reordering of fields in the record.
  - They allow changes to fields of the file.
  - They allow joins.

A logical file is comprised of two MI objects: a cursor and an optional data space index, as depicted in Figure 4-16. "Logical File Member Structure" on page 4-78.

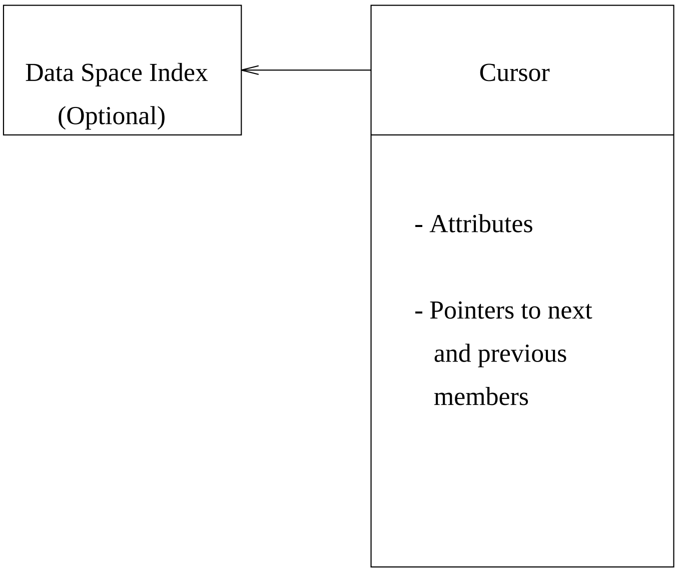


Figure 4-16. Logical File Member Structure

4.2.1.3.2. SLIC Database File Structure

The SLIC database provides the basic functions for creating, managing, and accessing database files. Several MI objects are associated with a database file. These objects include a data space, a cursor and an optional data space index (see section 3.2.1.5 “MI Objects” on page 3-26).

4.2.1.4. Data Organization

Data (or records) in a physical file is stored in data spaces. A data space is comprised of data segments where the database records are stored. These records (or data space entries) can be retrieved, updated, inserted or deleted. Entries in a data space are stored based on arrival order, with each entry having a permanently assigned order number identifier. Deleted entries occupy space unless the reuse attribute in the data space is specified.

A data segment is comprised of ‘non-data’ portion and data space entries (actual data or records), as shown in Figure 4-17. “Data Segment” on page 4-79. The non-data portion is comprised of the following entries:

- DENT byte: The DENT byte describes the current status of the data space entry (e.g., entry deleted). It is always the first byte of each entry in a data space.
- Auxiliary SID Relative Address (ASRA): This field is present only if variable length data is present in the data space. It contains the location of the variable length data.
- Auxiliary SID Entry Length (AxAL): This field is also present if variable length data is present in the data space. It contains the length of the variable length data.

- The NULL Bit Map: This field is present only if there are NULL capable fields in the data space entry. There is one bit for every field.
- The Length or Location Information (LORLI): This field is present at the start of each variable length field. If the high-order bit of this 2-byte field is ‘ON’, then the data for this field lies in the auxiliary SID. If the high order is ‘OFF’, the LORLI contains the length of the variable length data in the fixed length SID.

DENT	ASRA	AxAL	NULL	Fixed Length Field	LORLI	Variable Length Field	Record 1
DENT	ASRA	AxAL	NULL	Fixed Length Field	LORLI	Variable Length Field	Record 2
DENT	ASRA	AxAL	NULL	Fixed Length Field	LORLI	Variable Length Field	Record 3

⋮

DENT	ASRA	AxAL	NULL	Fixed Length Field	LORLI	Variable Length Field	Record N

Figure 4-17. Data Segment

4.2.1.5. Database Integrity

Data integrity protects data from being destroyed or changed by unauthorized subjects, hardware failure (such as physical damage to a disk), interruptions before a job is completed (such as a power failure), or interference from other applications running at the same time (such as serialization problems). Data integrity is ensured by the following functions:

- Concurrency control
- Atomic operations
- Journaling
- Commitment control
- Save/restore
- Damage tolerance
- Index recovery

Each of these functions are briefly discussed below. Refer to the section 4.3.2 “Backup and Recovery” on page 4-90 for more details.

4.2.1.5.1. Concurrency Control

Concurrency is the ability for multiple users to access and change data in the same table or view at the same time without risk of losing data integrity. Locks are required to protect concurrent users from changing the same data at precisely the same time. (Locks are discussed in details in section 3.2.1.6 “Locks” on page 3-38.)

The AS/400 DBMS has the capability to define the degree of isolation of an application process (i.e., job) from other concurrently executing processes. The isolation level of an application process X, therefore, specifies the following:

- The degree to which rows read and updated by X are available to other concurrently executing application processes
- The degree to which update activity of other concurrently executing application processes can affect X

The current evaluated configuration of AS/400 supports five isolation levels: Repeatable Read (RR), Read Stability (RS), Cursor Stability (CS), Uncommitted Read (UR), and No Commit (NC).

- **Repeatable Read**: RR ensures that any row read during a unit of work or UOK (a UOK is a recoverable sequence of operations within an application process) is not changed by other application process[27] that use different commitment definitions until the unit of work is complete. Any row changed (or a row that is currently locked with an UPDATE row lock) by another activation group using a different commitment definition cannot be read until it is committed.
- **Read Stability:** RS ensures that any row that is read during a unit of work is not changed by other application processes until the unit of work is complete. Any row that was changed by another application process cannot be read until it is committed by that application process. Unlike RR, RS does not completely isolate the application process from the effects of concurrent application processes. At level RS, application processes that issue the same query more than once might see additional rows. These additional rows are called phantom rows.
- **Cursor Stability:** Like RS, CS ensures that any row that was changed by another application process cannot be read until it is committed by the application process. Unlike RS, CS ensures only that the current row of every cursor is not changed by other application processes. Thus, the rows that were read during a UOK can be changed by other application processes.
- **Uncommitted Read:** UR allows any row that is read during the UOK to be changed by other application processes. Any row that was changed by another application process to be read even if the change has not been committed by that application process.
- **No Commit**: For all operations, the rules of level UR apply except that commit and rollback operations have no effect on SQL statements. Prepared statements are not discarded, cursors are not closed, and LOCK TABLE locks are not released. However, connections in the release-pending state are ended. Any changes are effectively committed at the end of each successful change operation and can be immediately accessed or changed by other application groups using different commitment definitions.

4.2.1.5.2. Atomic Operations

The AS/400 DBMS can ensure that most underlying database functions are atomic - either they will complete or they will appear to never have been started.

4.2.1.5.3. Journaling

The AS/400 journal support supplies a forward and backward recovery. Forward recovery can be used to take an older version of a table and apply the changes logged on the journal to the table. Backward recovery can be used to remove changes logged on the journal from the table.

[27] In the context of commitment control, the term *application process* means “an activation group that uses a separate commitment control definition.”

When a table is created into the collection, it is automatically journaled to the journal created in the collection. A user can stop journaling on any table by using the journal functions.

4.2.1.5.4. Commitment Control

Commitment control provides a means to process a group of database changes as a single unit of work (i.e., transaction). Commitment control guarantees the transaction is completed.

4.2.1.5.5. Save/Restore

The save/restore functions are used to save collections, tables, views, indexes, journals, and journal receivers on some external media or to a save file. All information needed to restore an object to its previous state is saved. Later, this information can be used to recover from damage to individual tables by restoring the data with a previous version of the table.

4.2.1.5.6. Damage Tolerance

The AS/400 DBMS has a certain amount of tolerance to damage caused by disk errors. In the event that a disk error has damaged a small portion of the rows in a table, the AS/400 DBMS allows use of the rows that are still accessible.

4.2.1.5.7. Index Recovery

The AS/400 DBMS provides several options to deal with index recovery. All indexes on the system have a maintenance option that specifies when an index is maintained. In addition, as part of IPL, database server systems jobs are started to perform index recovery.

4.2.1.6. Database Management System Transaction Processing

This section describes how the AS/400 manages access to database files by processing request statements (e.g., SQL statements, Query/400 statements, etc.).

The components that support the functionality of the DBMS implement generalized interfaces that satisfy requests from any number of components in the system (e.g., QM for Query Manager, SQ for SQL/400 Product, QU for Query/400). For example, the QQ component optimizes a query for any component on the system. Consequently, in this section, we describe two interfaces (SQL and application program) and their specific inputs and outputs.

The OS/400 database component[28] manages database files. There are two ways to open a database file: query open or simple open. A query open is used when data in a file is accessed and records are selected. A simple open is used when data in a database file is accessed without any additional selection criteria. Once the database file is open, I/O operations can be performed regardless of which open was used. An example of how a query is processed through a query open and a simple open from a database perspective is discussed next. For more information about generic database file opens, see section 4.1.1.1.2 “Data Management” on page 4-9.

[28] In the remainder of this chapter, we will refer to OS/400 database component as ODB and the SLIC database component as SDB.

4.2.1.6.1. Query Open

A query open is used when data in a database file is accessed with additional selection criteria (e.g., SELECT * FROM EMPLOYEE WHERE A > 1000). The following steps are performed as part of a query open (see Figure 4-18. "Query Open" on page 4-82):

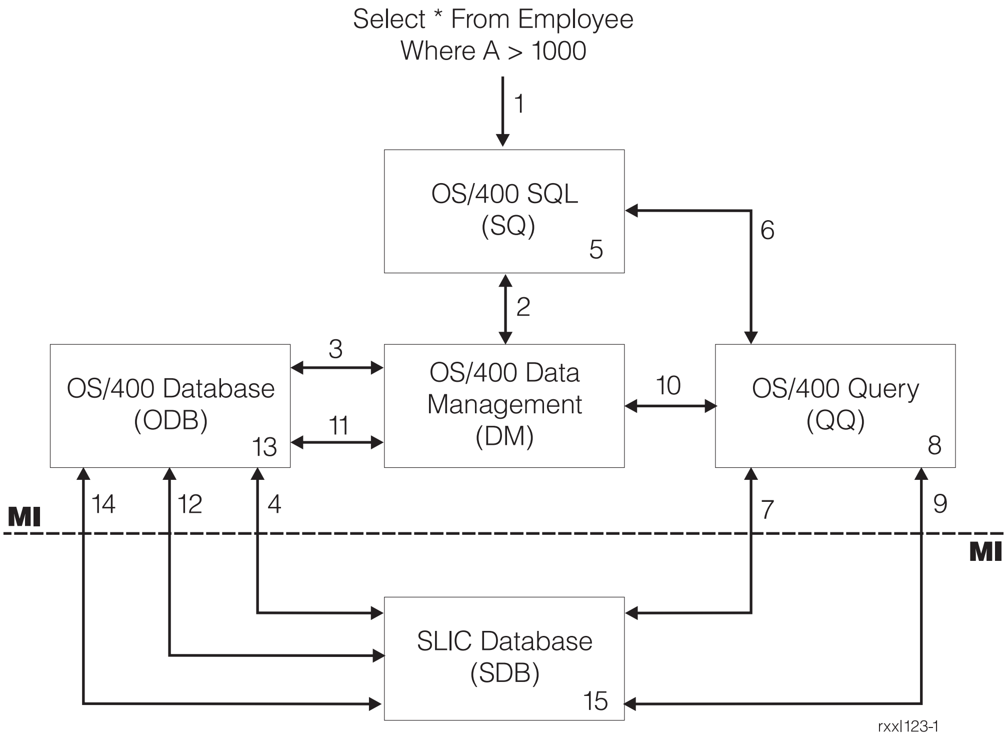


Figure 4-18. Query Open

1. The query is received by the SQ component. SQ parses the query and builds the Query Definition Template (QDT). The QDT contains the SQL tokens. SQL also builds the User File Control Block (UFCB). The UFCB is a structure that provides a single linking object to all other objects that are required to move data from a user to a database file or from a database file to the user. The UFCB is stored in user domain process local space. See section 4.1.1.1.2 "Data Management" on page 4-9 for more information how/where the UFCB is maintained. SQ then determines whether the query is correct.
2. SQ calls Data Management (DM) to extract information about the file (EMPLOYEE).
3. DM routes the request to ODB.[29]
4. ODB issues MI instructions to extract statistics about the file. Object Operational (*OBJOPR) authority is needed to extract the information. The statistics contain information such as size, number of records, field names, and domains. ODB returns this information to SQ.
5. SQ checks the semantics of the SQL statement. For example, it makes sure that NAME is a field in EMPLOYEE and that it is a character field.

[29] A key function of DM is to serve as a focal point for all file I/O requests. DM then routes the request to the appropriate component (e.g., ODB).

6. SQ calls QQ to optimize the query and do additional semantic checking. SQ passes QQ the QDT and the UFCB.

7. QQ validates the UFCB and performs authority checks by calling DB. DB checks whether the subject accessing a physical file has *OBJOPR authority and one or more data authorities. (If the subject is accessing a logical file, then DB checks whether the subject has *OBJOPR authority and one or more data authorities. If the subject is accessing a logical file, then DB checks whether the subject has *OBJOPR and one or more data rights to the logical file and one or more data rights to the underlying logical and physical file(s)). QQ then issues MI instructions to materialize statistics and definition information about indexes that may be available over the data space that it may decide to use in the query.

8. QQ determines the optimal way to process the query.

9. QQ issues the MI instruction Create Cursor (CRTCR) to create a temporary cursor. A temporary cursor, which is in user domain and process local write-protected storage, contains information about the query itself. The associated space of the temporary cursor, which is called the Open Data Path (ODP), is also in user domain and process local write-protected storage and contains information about I/O (e.g., feedback).

10. QQ passes the temporary cursor to DM, which registers the open in its control blocks (see section 4.1.1.1.2 “Data Management” on page 4-9).

11. DM then passes the temporary cursor to the ODB component.

12. ODB issues MI instructions to materialize statistics about data spaces (and data space index) that are part of the file. Object operational authority is required on this instruction. The statistics contain information such as number of records and number of fields. The ODB component stores this information in the ODP.

13. ODB allocates the buffers (input or output). The buffers are allocated in user domain storage. Data that is read from the file is returned to the user in the input buffer. Data that is provided by the user on updates is passed in the output buffer.

14. ODB then issues the MI instruction Activate Cursor (ACTCR) to complete the open.

15. SDB attaches the cursor, data space, and data space index (if used) to the activating process by storing the job identifier into the cursor. This action connects the cursor to the process, thereby restricting its use to the activating process and assuring that the cursor will be deactivated when the process terminates. SDB requires at least one data authority (*READ, *ADD, *DLT, or *UPD) to activate the cursor. The authorities that the user has to the file are now stored in the cursor to speed up the authority testing that then occurs at I/O time. The file is now open and I/O request may be performed. At this point, the file open is audited. The file being opened is always audited (if audit is turned on). For a logical file, the audit is also done for the underlying physical file(s). While a file is open, grants and revokes of authorities are prevented.

    Locks are acquired on the data space and the data space index by the MI instruction ACTCR and are held until the MI instruction Deactivate Cursor (DEACTCR) is executed, which is part of close processing. The lock acquired on the index is shared read (*SHRRD) to prevent it from being deleted, and the lock on the data space depends on how the file is opened and on the lock requested by the user. The default lock for read intent only is *SHRRD. The default lock for update, delete, or insert is shared update (*SHRUPD). This lock allows multiple concurrent updaters. Record locking at I/O time prevents conflicts at the record level.

4.2.1.6.2. Simple Open

A simple open is used when data in a file is accessed and selection is not performed on the records. The following steps are performed as part of a simple open of a database file initiated by an application program.

- A user domain user state program opens (language specific) a database file. At compile time, the compiler creates a data structure known as the User File Control Block (UFCB). The UFCB is a structure that provides a single linking object to all other objects that are required to move data from a user's program to a database file or from a database file to the user's program. See section 4.1.1.1.2 "Data Management" on page 4-9 for more information about how and where the UFCB is maintained.
- The compiler then generates an open call (a call to the QDMCOPEN UDSS program).
- At run time, the open call is routed to the Data Management (DM) component. The DM component serves as a common point for all file open requests.
- DM resolves the symbolic file name and issues MI instructions to extract information (e.g., file type) about the file. DM then performs authority checks to ensure that the subject has the proper authority to open the database file.
- When DM finds the database file, it issues the MI instruction Create Duplicate Object (CRTDOBJ) to create a temporary cursor that is a duplicate of the permanent cursor that is associated with the database file. The temporary cursor's associated space is known as the open data path or ODP. The temporary cursor is a user domain object, but it is created in write-protected storage. The UFCB is then updated with this information.
- DM calls the ODB component, passing it two parameters: the UFCB and the file open option (e.g., read, write).
- If the file is a logical file, ODB checks to ensure that the user has the appropriate data rights on the logical file and on each underlying logical or physical file.
- ODB issues the MI instruction to materialize statistics about data spaces (and data space index) that are part of the database file. Object operational authority is required on this instruction. The statistics contain information such as number of records and number of fields. The ODB component stores this information in the associated space of the temporary cursor (or ODP).
- ODB allocates the buffers (input and/or output). The buffers are allocated in user domain storage. Data that is read from the file is returned to the user in the input buffer. Data that is provided by the user on writes is passed in the output buffer.
- ODB then issues the MI instruction Activate Cursor (ACTCR) to complete the open.
- SDB attaches the cursor, data space, and data space index (if used) to the activating process by storing the job identifier into the cursor. This action connects the cursor to the process, thereby restricting its use to the activating process and assuring that the cursor will be deactivated when the process terminates. SDB requires at least one data authority (*READ, *ADD, *DLT, or *UPD) to activate the cursor. The authorities the user has to the file are now stored in the cursor to speed up the authority testing that then occurs at I/O time. The file is now open and I/O request may be performed. While a file is open, grants and revokes of authorities are prevented.

4.2.1.6.3. Database I/O

Once a file is open, I/O can be performed against it. For example, the following steps are performed as part of a get request:

- The get request is routed to the appropriate database get program.
- The ODB component issues the MI instruction Set Cursor (SETCR) to position the cursor to the appropriate record and to lock the record.
- The ODB component then issues the MI instruction Retrieve Sequential Data Space Entries (RETSDSE) or Retrieve Data Space Entry (RETDSEN) to retrieve the data associated with the current cursor position.
- The SDB component verifies that the user has *READ authority and returns the record to ODB.
- ODB sets I/O feedback information into the temporary cursor (ODP) and returns to the application program.

4.3. Supporting Functional Areas

4.3.1. Initial Program Load (IPL) and Installation

This section discusses how operators power on the system and is followed by a thorough discussion of IPL. This includes how IPL is initiated, what options are available during an IPL, what occurs during an IPL, installing OS/400, and reaching the C2 configuration. The section closes with a discussion of shutdown.

4.3.1.1. System Power-On

There are several ways in which the system may be powered on:

- Manual power-on (using the Power switch)
- Service processor power-on (timed power-on or automatic power restoration)
- Remote power-on (disallowed in the C2 evaluated configuration and not discussed further)

Manual power-on: Allows an authorized user to power on the system using the Power switch on the control panel. The manual power-on powers on the system only when the keylock switch is in the manual or normal position.

Service processor power-on: The service processor can initiate a system power-on provided that the keylock switch on the control panel is in the auto or normal position. With power off, the service processor can initiate a power-on at a time and date as specified by a system value. The service processor can initiate an automatic power restoration after an abnormal power-off (e.g., a power failure).

An IPL is performed after initiating any type of system power-on.

4.3.1.2. IPL Initiation

IPL is initiated by the occurrence of several system events or internal conditions:

- By a system power-on as described above in section 4.3.1.1 “System Power-On” on page 4-85
- Selecting a control panel function (that starts an IPL and loads the system)
- Executing a programmed IPL (issuing the CL command Power Down System (PWRDWNSYS) with the RESTART(*YES) parameter)
- Accepting a Unit Reset-D Command (on a multiple I/O bus configuration).

4.3.1.3. IPL Options

An IPL is done in either attended or unattended mode. Authorized users who perform attended mode IPLs are presented with displays (which include the Dedicated Service Tools (DST) display and Sign On display) on the console during IPL. In unattended mode, the operator does not interact with the console.

The Sign On display appears only during attended mode IPLs. A request is made at the Sign On display for the operator to present a user profile and password, as would be done at a normal terminal login. If there are problems during the attended mode IPL, such as the console not functioning, then the attended mode IPL switches to an unattended mode IPL. The Sign On display does not appear on any terminals during an unattended mode IPL until IPL is complete.

The position of the keylock switch on the control panel influences the type of IPL that can be performed. The keylock switch has four positions:

Manual: When the keylock switch is in the Manual position, an attended mode IPL is always attempted by the system. An attended mode IPL is performed by moving the power switch on the control panel from Delayed Off to the Power On position. An attended mode IPL is also performed by selecting a control panel function (that starts an IPL and loads the system). Timed IPLs and automatic IPLs cannot be performed.

Secure: When the keylock switch on the control panel is in the Secure position, IPLs can occur only when an authorized operator issues a programmed IPL (issuing the CL command PWRDWNSYS with the RESTART(*YES) parameter). Timed IPLs and automatic IPLs cannot be performed.

Normal: The following occurs when the user places the keylock switch in the Normal position. An IPL will be performed by moving the power switch on the control panel from Delayed Off to the Power On position. Operators cannot select control panel functions to begin an IPL. Timed IPLs and automatic IPLs can be performed. IPLs can be performed in either attended or unattended mode, depending on a system value.

Auto: Placing the keylock switch on the control panel in the Auto position allows timed IPLs and automatic IPLs. IPLs can be performed in either attended or unattended mode, depending on a system value.

A programmed IPL occurs after issuing the CL command PWRDWNSYS with the RESTART(*YES) parameter. Programmed IPLs may be performed by an authorized user when the keylock switch is in any of the four positions:

- Programmed IPLs with the keylock switch in the Manual and Secure positions will be performed in attended mode.
- Programmed IPLs with the keylock switch in the Normal and Auto positions will be performed in either attended or unattended mode depending on a system value.

When performing an IPL or an installation, a copy of the SLIC must be placed into the system. There are four separate copies of SLIC:

- A – IPL from disk using copy A of the SLIC
- B – IPL from disk using copy B of the SLIC
- C – Reserved for IBM
- D – IPL from tape. Used for code installation and restoration of the SLIC

The system maintains two copies of SLIC on the system. Copy A is considered the permanent copy, while copy B is considered the temporary copy because Temporary Program Temporary Fixes (PTFs) can be applied to this copy. Operators performing an IPL may use copy A or copy B of the SLIC if all PTFs have been applied permanently. If PTFs have been applied, then copy A of the SLIC must be used to perform IPLs. Operators performing an installation (which are always done from tape) must use copy D to load SLIC.

4.3.1.4. What Occurs During an IPL

IPLs are performed to load system programs from auxiliary storage, test system hardware, and free memory.

Once initiated, an IPL proceeds through four phases:

1. Phase 1 - Service Processor

   The following is a general overview of the functions performed by the service processor during IPL.

   - The service processor IOP becomes the Bus Control Unit (BCU) and performs basic assurance tests on the control panel, service processor, and every Input/output Processor (IOP) (see section 3.1.1.2.2.3 “Service Processor” on page 3-14).
   - The Read Only Storage (ROS) of the service processor loads its functional code from the load source DASD. It then reads the Central Electronic Complex’s (CEC) Vital Product Data (VPD) to determine what additional code loads are needed by itself and the CEC.
   - The service processor then loads the processor-dependent code and executes the CEC hardware diagnostics and initialization functions. It builds system VPD and stores it in an architected area of main storage that SLIC can access.
   - The service processor loads the SLIC nucleus into main storage and starts the processor. This step causes the SLIC initialization tasks that reside on the task dispatching queue to be started. The completion of this step represents the start of the SLIC initialization phase.

2. Phase 2 - SLIC Initialization

   The following is a high-level list of the functions performed by SLIC during IPL.

   - SLIC sets the internal processing unit registers and arrays necessary to run SLIC. When this initialization is complete, SLIC signals the service processor of the event. SLIC then enters the task dispatcher disabled state. The service processor starts the processor and the previously constructed dispatcher queue is made ready. SLIC is now in control of the system.
   - Storage management's initialization code is run. The bus manager initializes itself and the load source bus unit is identified. Next, the load source DASD unit is started. Initialization code is run for Auxiliary Storage Management (ASM) and System Auxiliary Storage Pool (ASP).
   - All other bus units on all buses are loaded. Machine Initialization Status Record (MISR) initialization code is run. Resource Management initialization code is run. Exception Management initialization code is run. Process Management initialization code is run.
   - SLIC collects VPD for all of the attached IOPs and devices and creates a hardware driver for each IOP. The hardware drivers build internal hardware resource information structures.
   - The Dedicated Service Tools (DST) task is created, which creates the initial machine process.
   - SLIC is now fully operational.

3. Phase 3 - Machine-to-Program Transition

   During this phase, a determination is made whether an installation of OS/400 or an IPL of OS/400 is to be performed.

4. Phase 4 - OS/400 Installed and/or Started via IPL

   OS/400 is either installed or started in this phase. OS/400 performs it's initialization and presents the sign-on screen.

4.3.1.5. Reaching a C2 Configuration

AS/400 arrives at a site with the SLIC and OS/400 already loaded on the system. To reach the C2 configuration, the following events must occur during the first on-site installation of AS/400:

1. A user capable of issuing the CL command PWRDWNSYS must log on.
2. The system must then be placed in Manual mode using the control panel on the system unit.
3. Before installing a new copy of SLIC and the OS/400 operating system from tape (or optical disk), the user must erase all information from all disk units on AS/400.
4. After the shipped SLIC is replaced, the OS/400 is now installed.
5. After OS/400 has been installed, the system values pertaining to the C2 level of security must be set to the values indicated per the installation instructions in the Guide to Enabling C2 Security [10].
6. The security officer must change his or her password, and then the security officer must log in using the new password.
7. At this point, the optional parts of the operating system and additional licensed programs that have been evaluated to meet the C2 requirements may now be added. The security officer must apply feature code 1920, which includes a cumulative PTF package required to meet C2 requirements.
8. The system must now be placed in Secure mode. The key must then be removed and secured by the security officer of the system.

9. After installing the operating system and licensed programs on AS/400, auditing may be started on the AS/400.
10. Finally, there are a set of C2 customization tool programs that must be run. These set the public authority to *EXCLUDE for objects and interfaces not to be used in the evaluated configuration.

4.3.1.6. System Shutdown

AS/400 users with *ALLOBJ and *JOBCTL special authorities can take the AS/400 out of the evaluated configuration by issuing one of the following CL commands:

- The CL command End System (ENDSYS)
- The CL command End Subsystem (ENDSBS) with the SBS(*ALL) parameter
- The CL command Power Down System (PWRDWNSYS)

The CL command ENDSYS and the CL command ENDSBS SBS(*ALL) place the machine in a restricted state such that it is usable only from the console. The CL command PWRDWNSYS brings the power completely down. Each brings AS/400 out of operational mode.

The following is the process of bringing the system out of operational mode.

```
Authorized User --> PWRDWNSYS or ENDSYS or ENDSBS SBS(*ALL)
                         |
                         |-->Signals system arbiter
                             |   |
                             |   |-->Terminates subsystems
                             |       |
                             |       |->Terminates jobs
            (The CL command ENDSYS and ENDSBS SBS(*ALL) stop here)
                             |
                             |-->Signals Start CPF
                                 |   |
                                 |   |-->Ends system jobs
                                 |   |-->Closes files and I/O
                                 |
                                 |-->Terminates machine process
                                     |
                                     |->Power off
```

Both the CL command PWRDWNSYS and the CL command ENDSYS call the same CPP. To distinguish one command from the other, each command sends different parameters to the CPP being called. Each of the three CL commands performs clean-up and ends jobs and subsystems in the same way.

The process of bringing the system out of operational mode begins by verifying that the requester is authorized to the command. If the CL command ENDSYS or ENDSBS SBS(*ALL) are issued, they must be issued from the controlling subsystem. The CPP then signals an event to the system arbiter to begin shutting down the subsystems.

An event handler in the system arbiter is given control and begins sending terminate events to the active subsystems. Before the subsystems are terminated, each subsystem blocks any new jobs from starting and terminates the currently running jobs. This happens in all subsystems except the controlling subsystem when either the CL command ENDSYS or the CL command ENDSBS SBS(*ALL) is issued. Taking the system out of operational mode can be controlled or immediate. If it is controlled, a timer is set and messages are sent to notify active jobs that the system is terminating. If it is immediate, the work management component starts the job termination after clean-up. When all of the jobs have been terminated, the system is out of the evaluated configuration.

As each subsystem is terminated, an event is signaled that removes the subsystem from the system control block chain. If either the CL command ENDSYS or ENDSBS SBS(*ALL) is issued, an event in the controlling subsystem is given control when the system reaches the restricted state. The CL command ENDSYS and ENDSBS SBS(*ALL) are finished at this point.

If the CL command PWRDWNSYS was issued, the event that has control in the controlling subsystem signals the power-down notification event, calls device configuration to quiesce the I/O devices, and closes any open data base files.

At this point the Start CPF Program takes over, and the system arbiter process terminates. All system jobs are terminated and machine processing ends. The SLIC then gets control, runs shutdown code for its tasks, sets the service processor code, and issues an instruction telling the processor to shut the power down.

4.3.2. Backup and Recovery

Backup and recovery for AS/400 is accomplished by the Save/Restore (SR) OS/400 component and the Load/Dump (LD) component of the SLIC, both of which run in system state. The SR job above the MI and the LD tasks below the MI provide a way to back up and recover objects and to exchange objects between systems. SR and LD are discussed in this report for the following three reasons:

- Restoring a program object (i.e., an object of type *PGM or *SRVPGM, i.e., an executable program) containing MI instructions from a non-C2 AS/400 system could allow the use of MI instructions that are blocked at translate time but not at run time on the evaluated configuration. Such a program could be written on a non-C2 AS/400 system, using a third-party translator to run in system state. Using Restore to put such a program on a C2 AS/400 system would violate the system architecture requirement.
- If objects could be backed up and then restored without the original authorities attached, DAC could be violated.
- If deleted objects can be restored improperly, the object reuse requirement could be violated. (“Improperly” here means without the original owner or authorities, or perhaps against the original owner’s intentions.)

4.3.2.1. Authorities and Special Authorities for Doing Saves and Restores

In a C2 configuration, system administrators set public authority to *EXCLUDE for all save and restore CL commands. In addition to having authority to the CL command, the user must have authority to the object that is the target of the save or restore CL command. Two ways exist to establish authority to save or restore the object:

- Having the *OBJEXIST authority to the object. A user profile must have *OBJEXIST to save and restore objects. See section section 6.2 “Discretionary Access Control” on page 6-4 for more about *OBJEXIST.
- Having the *SAVSYS special authority. A user profile holding the *SAVSYS special authority can use the various save and restore CL commands for all objects on the system, regardless of whether that user profile has object existence (*OBJEXIST) authority for the object. The holder of *SAVSYS special authority needs no additional authority to the object to save or restore it. A user profile of security officer or system operator user class will have the *SAVSYS special authority. See section 5.2.6 “User Classes and Special Authorities” on page 5-4 for more about user classes.

In addition to the special authority or object authority to use the save or restore CL command and authority to save or restore the target object, the user profile doing a save must have *USE authority to the tape or diskette unit being used as the save media. If the save is done to a save file on the system, the user profile needs *ADD if the save file is empty, plus *ADD and *OBJMGT if the save file has existing records in it. Also, the user profile must have *EXECUTE to the library containing the object. For restores, the user profile needs *ADD (for objects not already existing in the library) plus *EXECUTE authority (for objects already existing in the library) on the library into which the object is restored.

4.3.2.2. Save

From the perspective of the SR job at the OS/400 level, the objects making up the AS/400 evaluated configuration are divided into eight categories. These categories do not correspond to the object types defined in section 5.3 “Objects” on page 5-5. Instead, the groupings are particular to the SR component and the various save and restore CL commands. The eight categories are as follows:

- SLIC
- Device Configuration Objects, such as controller, and device descriptions and other system resource management objects
- User Profiles
- OS/400 Library QSYS, which is owned by the user profile QSYS. The system values reside in library QSYS.
- Libraries for Licensed Programs, as listed in Appendix B. Examples include SQL/400 and QRY/400.
- QGPL and QUSRSYS, two libraries that typically contain objects such as journals and journal receivers, output queues, and message queues, but which can contain other user objects.
- All other objects not covered above

The various save CL commands save all or some of the objects listed above. Saving and restoring security information in particular depends on the commands chosen; see Table 4-11 on page 4-91 for details on saving and restoring security information only.

Table 4-11. Saving and Restoring Security Information

Data Saved or Restored	**SAVSYS** **SAVSECDTA**	**SAVCHGOBJ** **SAVOBJ** **SAVLIB** **SAVDLO** **SAVCFG** **SAV**	**RSTUSRPRF**	**RSTOBJ** **RSTLIB** **RSTDLO** **RSTCFG** **RST**	**RSTAUT**
User Profiles	X		X		
Object Ownership (not user profiles or auth. lists)		X		X	

Table 4-11. Saving and Restoring Security Information

Data Saved or Restored	**SAVSYS** **SAVSECDTA**	**SAVCHGOBJ** **SAVOBJ** **SAVLIB** **SAVDLO** **SAVCFG** **SAV**	**RSTUSRPRF**	**RSTOBJ** **RSTLIB** **RSTDLO** **RSTCFG** **RST**	**RSTAUT**
Object Ownership (for user profiles and auth lists)	X		X		
Public Authorities (Not user profiles or auth. lists)		X		X	
Public Authorities (for user profiles and auth lists)	X		X		
Private Authorities (other than owner's private authority)	X				X
Authorization Lists	X		X		
Secured by Auth List		X		X	
Object Auditing Value (not saved or restored by SAVCFG or RSTCFG)		X		X	

When user profiles are saved, the private authority information stored with the user profile is formatted into an *authority table*. This is done for each user profile that has private authorities. All other objects are saved intact.

4.3.2.3. Restore

There are restore CL commands corresponding to the save CL commands; those restore CL commands that handle security information are shown in Table 4-11 on page 4-91.

Objects that were saved by separate commands must be restored by separate commands. Objects that do not exist on the system can be restored. When an existing object is restored, the restored object is copied over the existing object.

At the OS/400 level, one restore CL command is used to restore Licensed Products (as listed in Appendix B) when the restore is not part of a full system install. In the evaluated configuration, only users with the *ALLOBJ special authority can use this command because this CL command is shipped with Public set to *EXCLUDE.[30]

The restore CL command that restores user profiles also restores authorization lists. Authorization lists are reattached to the objects when the object is restored unless the object is in QSYS. An IBM-supplied program must be used to attach the authorization lists for objects in QSYS; these objects include the CL commands and other system-state programs. Private authorities for objects owned by other users are not restored by this restore CL command; instead, a specific restore command for private authorities must be used.

For restores of objects secured by authorization lists, the following rules apply:

- When an object already exists on the system and is linked to an authorization list, the authorization list must have the same name and have been saved from the same system as that for the saved object. If the name of the authorization list matches and the object is being restored on the same system from which it was saved, the object is linked to the authorization list again. If not, the object is not restored.
- If an authorization list of that name does not exist on the system, the object is restored without being secured by an authorization list and the public authority is changed to *EXCLUDE.
- If the object is being restored on a different system, the object is not linked to the authorization list (and public authority will be set to *EXCLUDE) unless the ALWOBJDIF parameter on the restore CL command is set to *ALL (*ALL or *AUTL for objects in directories). The special authority *ALLOBJ is required to use the ALWOBJDIF parameter on a restore CL command.

When private authorities are restored by the above-mentioned specific restore CL command, the command rebuilds the private authority in the user profile using the information from the authority table at the time the user profile was saved. The grant authority operation is run for each private authority in the authority table. If existing user profiles have private authority to an object being restored, those private authorities are not affected unless the object is a program. (See below.) When an object is deleted, all private authority to the object is removed from user profiles. If an object is deleted from the system and then restored from a saved version, private authority for the object no longer exists on the system and must be restored with the specific private authority restore CL command.

If user profiles are restored that did not exist on the system before, the current password (kept in the AUT) is set to *NONE. In addition, *ALLOBJ special authority is removed from user profiles when these profiles are being restored from a different system or from the same system when it was not in the evaluated configuration.[31] However, *ALLOBJ authority is not removed from the IBM-supplied user profiles QSYS, QSECOFR, QLPAUTO, or QLPINSTALL.

[30] The TFM recommends that this CL command be run from a program that adopts *ALLOBJ special authority and then drops the adopted authority, instead of running the CL command directly. In this way, the user will not require *ALLOBJ special authority. This recommendation is made because this particular restore CL command calls an exit program to install programs that are not IBM-supplied. (See section 4.1.3.14 “Exit Points, Exit Programs, and User Exits” on page 4-67 for more about exit programs.) Using adopted authority as outlined in the TFM ensures that the adopted authority is dropped when the exit program is called.

[31] The system uses the machine serial number on the system and on the save media to determine whether objects are being restored to the same or different systems. It uses the QSECURITY value to determine whether the saved system was in the evaluated configuration.

When objects are restored, object ownership is also restored according to the following rules:

- If the user profile for the saved object matches the owning user profile on the system, ownership is restored to that profile.
- If the owner profile does not exist on the system, ownership of the object is given to the QDFTOWN (default owner) user profile.
- If the object exists on the system and the owner of the object on the system is different from the owner on the save media, the object is not restored unless the ALWOBJDIF parameter on the restore command is set to *ALL (*ALL or *OWNER for objects in directories). In that case, the object is restored and the owner on the system is used.

When public authority is restored, the following rules are followed:

- If the object being restored does not exist on the system, public authority is set to the public authority of the saved object.
- If the object being restored does exist and is being replaced, existing public authority is not changed; the public authority from the saved version of the object is not used.

When object auditing is restored, the following rules are followed:

- If an object being restored does not exist on the system, the object auditing (OBJAUD) value of the saved object is restored.
- If the object being restored does exist on the system and is being replaced, the object auditing value is not changed. The OBJAUD value of the saved version of the object is not restored.
- If a library being restored does not exist on the system, the create object auditing (CRTOBJAUD) value for the saved library is restored.
- If a library being restored exists and is being replaced, the CRTOBJAUD value for the saved library is not restored. Instead, the CRTOBJAUD value for the existing library is used.

The evaluated configuration restricts user domain objects of types *USRSPC, *USRIDX, and *USRQ to the library QTEMP. If an object of these three types is restored to any other library, the restore procedure changes the object to system domain when it is restored.

4.3.2.4. Validation of Programs Being Restored

The restoration of all objects to the system can be audited. The TFM directs the security officer to monitor the audit trail for object restoration audit records in which the object type is *PGM. In addition, a technique called *program validation* is used to determine whether an object of type *PGM being restored to a system in the evaluated configuration should in fact be restored without further procedural checks being made by the system administrator. If further checks are needed, the restoration process can include writing an audit record to record an authority failure when the validation check fails, and can include changing ownership of the *PGM object. The TFM directs the security officer to monitor the audit trail for these authority failure audit records also, and tells the security officer how to check if the restored program is a system state program by using the audit information. The procedure for checking the audit trail is discussed further below.

Program validation must be done on the evaluated configuration because the *PGM object may contain MI instructions that would be blocked at translation time in the evaluated configuration.[32] In addition, the *PGM object may run in system state. If the restore is allowed, the *PGM object allows violation of the sys-

[32] Section 4.1.1.1.1 “Program Management” on page 4-8 has described the structure of a *PGM object. There are no interfaces to restore any piece of executable code (for example, an executable RISC program) not in the form of a *PGM object.

tem architecture requirement. (See section 7.1 “System Architecture” on page 7-1 for more about how the system architecture requirement is supported by the evaluated configuration.) Such a *PGM object could result from the object being translated on a system not in the evaluated configuration, or from the object code being altered by use of service tools that are not in the evaluated configuration.

Program validation is done using the validation value. When a program is created on the evaluated configuration, the Translator calculates a validation value, or checksum, which is stored with the program. When the program is restored, the validation value is calculated again by the object-specific Load/Dump (LD) task for the *PGM object and compared to the validation value that is stored with the program.

Figure 4-19. “Retranslation Decisions” on page 4-96 shows the decision process for program validation. All branches shown in the figure are valid for the evaluated configuration. This decision process is used whether an object of type *PGM is being restored individually, or as part of a wider restore that includes one or more objects of type *PGM. The decision path depends in part on the setting of a parameter as part of the restore CL command. This parameter is ALWOBJDIF, which specifies whether differences encountered during a restore operation are allowed. One such difference is object ownership of the object on the system and on the save media. Another such difference is the validation value. The settings of the ALWOBJDIF parameter and the effect the setting has on processing, are described below. When this parameter is used, the user must have *ALLOBJ authority to the restore CL command.[33]

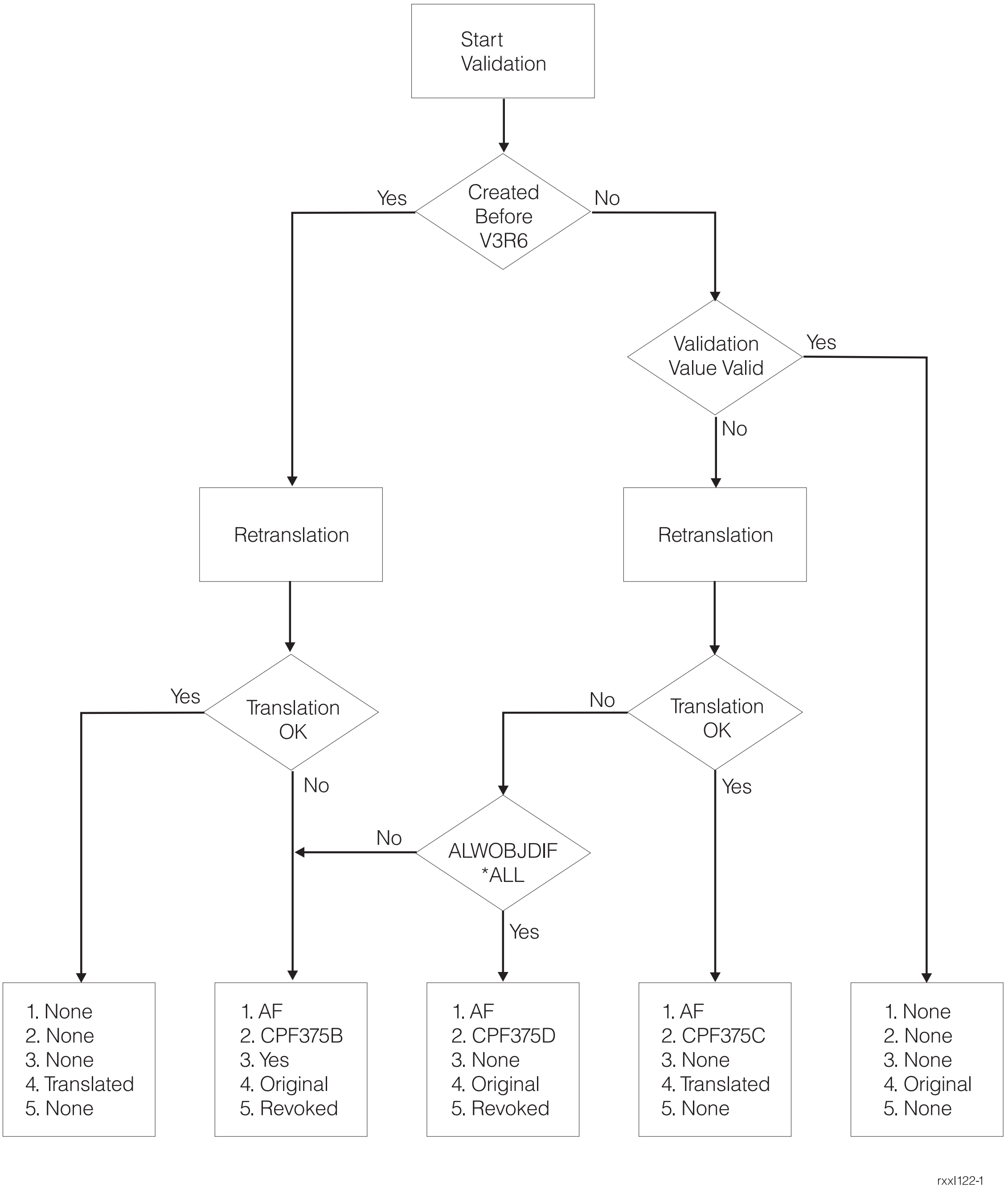
As Figure 4-19. “Retranslation Decisions” on page 4-96 shows, the first decision is whether the *PGM object was created before OS/400 Version 3 Release 6 (V3R6). This distinction is made because programs created before V3R6 contain IMPI instruction streams rather than RISC instruction streams.

The left branch in Figure 4-19. “Retranslation Decisions” on page 4-96 shows that a *PGM object created before V3R6 is retranslated. After retranslation, the retranslated *PGM object is compared to the restored *PGM object by the restoration process. If they match, then the retranslated *PGM object is restored, no authority failure audit records are written, and no changes in ownership or authorities are made. If they do not match, then an authority failure audit record is cut, an entry is made in the job log, ownership is changed to a default owner, and the authorities to the *PGM object are revoked.

The right branch also shows what happens to *PGM objects created for V3R6 or later. When the validation values do not match, a retranslation is forced by the restoration process and the restored and retranslated are compared. If they match, an authority failure audit record is written, an entry is made in the job log, and the retranslated version is restored. No changes are made to ownership or authorities. The authority failure audit record alerts the security officer to investigate whether the *PGM object has been modified in some way. If the two versions do not match, the setting of ALWOBJDIF is tested. If it is set to *ALL, an authority failure audit record is written, an entry is made to the job log, no changes are made to ownership, the *PGM object from the save media is restored, and authorities to the *PGM object are revoked. If ALWOBJDIF is not set to *ALL, the same actions are taken by the restore process except that ownership is changed to a default owner.

When a program is restored that adopts owner authority, the ownership and authority to the program may be changed. The following rules apply:

[33] The tests of ALWOBJDIF are for the setting of *ALL. If the user does not have authority to use the parameter, the tests fail and the effect is as if the parameter was set to *NONE.



System Action:

1. Entry in QAUDJRN
2. Message to JOBLOG
3. Changes in Ownership
4. Program Version Restored
5. Changes in Authority

Figure 4-19. Retranslation Decisions

The user profile doing the restore operation can receive the authority to restore the program by doing the following:

- Being the program owner
- Being a member of the group profile that owns the program (unless the user profile doing the restore operation has private authority to the program[34]
- Having *ALLOBJ and *SECADM special authorities
- Being a member of a group profile that has *ALLOBJ and *SECADM special authorities
- Running under adopted authority that meets one of the tests listed above
- If the restoring profile does not have adequate authority, all private authorities to the restored program are revoked, and the public authority is changed to *EXCLUDE.
- If the owner of the program does not exist on the system, ownership is given to the QDFTOWN user profile. Public authority is changed to *EXCLUDE and the link to any authorization list for the object is removed.

Once the restore process has completed, the system administrator or security officer can use the manual procedures detailed in the TFM to provide further protection against the restoration of a program object that breaks the evaluated configuration. The TFM directs system administrators to examine the audit entries for any restores of objects of type *PGM.

In addition, the TFM directs the system administrator to look for authority failure (shown as AF in Figure 4-19. “Retranslation Decisions” on page 4-96) audit records that show restoration of a *PGM object that failed the validation check. It further directs the system administrator to determine whether the restored program runs in system state. (The audit record for the *PGM restore contains this information.) If so, the TFM directs the system administrator to determine the program’s origin and to re-create the program, if possible, to determine whether it has been altered.[35]

Audit evidence for restored commands (objects of type *CMD) is also to be checked. The TFM directs the system administrator to check to see if the *CMD object calls *PGM objects that run in system state by using a CL command.

4.3.2.5. System Value - QALWOBJRST

The QALWOBJRST system value determines whether objects that are security-sensitive may be restored. The value can be set to *ALL, *NONE, *ALWSYSSTT or *ALWPGMADP. The user may specify *ALWSYSSTT and *ALWPGMADP at the same time.

- If the value is *ALL, any object may be restored to a C2 system by a user with the authority to the restore commands.
- If the value is *NONE, security-sensitive objects, such as system state programs or programs that adopt authority, may not be restored to the system.
- If the value is *ALWSYSSTT, system state objects may be restored to the system.
- If the values is *ALWPGMADP, objects that adopt authority may be restored to the system.

[34] The assumption here is that the user profile, although a member of the group, could have been explicitly given fewer rights than the group owner rights, and therefore user profiles with private authority to the program are excluded from this category.

[35] Use of the CL command Change Program with the Force Create parameter, as detailed in a previous footnote, is used to do this.

Note: The QALWOBJRST value must be set to *ALL before performing some system activities:

- Installing a new release of the OS/400 licensed program
- Installing new licensed programs
- Applying PTFS
- Recovering your system

The TFM states that a value of *NONE is required for the QALWOBJRST system value to prevent programs that adopt or run in system state from being restored on to the system.

4.3.3. Source Entry Utility

The SEU is a full screen editor that allows users to create, delete, and manipulate source physical files (e.g., create CL source files –*FILES).

This editor does not enforce any of the security policies; it requires the underlying OS/400 and SLIC to make the appropriate access checks and generate audit records when users open or save files with this utility. SEU also provides the user with the capability to execute CL commands, but performs no access checking on the command; access checking is performed by the command analyzer.

SEU runs in system state to directly invoke SDSS programs in the underlying OS/400 and could, therefore, possibly circumvent one or more of the security policies enforced by the system. It is for this reason that this product is considered part of the TCB.

Section 5
Trusted Computing Base (TCB) Protected Resources

This section discusses subjects and objects, the protected resources of the AS/400. It also describes the access control policy enforced by AS/400.

5.1. Users

A user, per the Trusted Computer System Evaluation Criteria (TCSEC), is any person who interacts directly with a computer system. A user can interact with an AS/400 in one of two ways:

1. The user must successfully sign on the system.

   To sign on, a user must have an account. The account is represented by an instantiation of a named object called a user profile. (See section 4.1.1.1 “External Objects” on page 4-2 and section 5.3.3.1.1 “User Profiles” on page 5-15). To validate the user to the account, a password is required (see section 6.1 “Identification and Authentication” on page 6-1).

2. An anonymous user can interact with the system by accessing a workstation that is under the control of an application started by a user[1] who has already successfully signed on (identified user).[2]

   The anonymous user is logically the guest of the identified user associated with the application, and the identified user is responsible (individually accountable) for all actions performed by the anonymous user. The application controls and limits which functions are available to anonymous users of the workstation. The application can use validation lists (*VLDL) to provide its own user identification functions; however, the system treats all anonymous users of the application as being the identified user associated with the application (including DAC checks and auditing).

User accounts, i.e., user profiles, are established as either trusted (administrators) or untrusted (other users). Administrators are role-based users that have been granted one or more special authorities that allow them to perform privileged functions. All users are members of *PUBLIC. A set of user profiles is furnished by IBM with the system. They are identified in Appendix I, “IBM-Supplied User Profiles”, on page I-1.

[1] The application can be started by any means supported by the system. For example, the signed-on user can call the application from a workstation. Alternatively, the signed-on user can submit a job that will run the application in another subsystem. In this case, as well as all other cases when the application can be run in an environment separate from the signed-on user’s job, an “identified user” is associated with the application (through a user profile) and is individually accountable for the actions of the application.

[2] An example of this type of application would be a survey program that solicits feedback from people who view an exhibit or participate in an event. The application limits the user’s interactions with the system to those functions it was designed to perform (e.g., updating the survey database files but providing no access to any other database files). The anonymous user cannot freely choose to start arbitrary programs, cannot freely choose to access arbitrary database files, and in general has no direct control over which parts of the system he or she interacts with.

An individual user may have more than one account. This is recommended in the Trusted Facility Manual (TFM) for administrators who may need to possess various authorities to perform various tasks. The object owner and private authorities of a user are contained in the user profile. When a user signs on an AS/400 system (see section 6.1.1 “Sign-on Process” on page 6-1), the name must identify a valid user profile. The user profile then maps the user to a subject. All subjects represent an individual user; however, one user may simultaneously have multiple subjects. The TFM recommends that users be restricted via a system value to being signed on at only one workstation at any given time.

The AS/400 has the concept of group profiles (see section 5.3.3.1.2 “Group Profiles” on page 5-19). The group profiles are an easy way to manage grouping of authorities. The TFM states that group profiles should be set with password *NONE, preventing users from signing on as a group. The assigning of the individual user to one or more user profiles ensures auditing to the individual user level.

5.2. Subjects

This section discusses the creation, deletion, and security attributes of AS/400 subjects. It also discusses *trusted subjects* in AS/400.

At the OS/400 level, a subject is the active phase (Phase 2) of a job. At the SLIC level, a subject is a process, which is the active phase of a job. Section 4.1.3.3 “Job Management” on page 4-23 describes job management in detail, and section 3.2.3.4 “Process Management” on page 3-61 discusses process management in detail.

Access checks to objects are made based on the job (process) that is running on behalf of a user.

5.2.1. Subject Attributes

There are several security attributes associated with a subject. These attributes are stored in several data structures. These structures and their associated attributes are as follows:

Process Control Space (PCS): Each subject in the system has associated with it a PCS Machine Interface (MI) object. Section 3.2.3.4 “Process Management” on page 3-61 describes the management of this object. The PCS contains security relevant information; such as, process state and a pointer to the User Profile on whose behalf the process is executing.

Invocation Control Block (ICB): The ICB is associated with a program called by a process. The ICB contains security relevant information, such as state transition and adoption status.

User Profile: The user profile associated with a subject contains several security attributes related to the subject. This object is described in detail in section 5.3.3.1.1 “User Profiles” on page 5-15.

Group Profile: The group profile is a user profile object that contains several security attributes for any subject contained in the group represented by that group profile. Group profiles are described in section 5.3.3.1.2 “Group Profiles” on page 5-19.

The use of subject attributes in determining whether the subject will be granted access to an object are discussed in section 6.2 “Discretionary Access Control” on page 6-4. The use of subject attributes in determining whether the subject may invoke a special privileged MI instruction is described in section 3.2.2.2.4 “Special Privileged Instructions” on page 3-41. The use of subject attributes in determining events to be audited is discussed in section 6.5 “Audit” on page 6-18.

5.2.2. Subject Creation

As described in section 3.2.3.4 “Process Management” on page 3-61, a subject is created when the MI instruction Initiate Process (INITPR) is invoked.

A user can cause a subject to be created on his or her behalf by interactively logging on the system or by submitting a batch job. When a user logs on, a subject is created after successful Identification and Authentication (IA) (see section 6.1 “Identification and Authentication” on page 6-1). When a user submits a batch job, it is placed on a job queue and a subject is subsequently created (see section 4.1.3.3.4 “Batch Processing Jobs” on page 4-32).

Some subjects are also created as part of system initialization (e.g., the QSPLMAINT job). Although these subjects run on behalf of IBM-supplied user profiles that do not represent users, their access to objects is still controlled based on the subject identity.

Subjects may also be created using CL commands. For example, the the CL command STRPRTWTR causes a spooling writer job to start, which runs under the user profile QSPLJOB (see section 4.1.3.4.4.4 “Spooling Writers” on page 4-45). Similar to the subjects created at system initialization, these subjects run on behalf of IBM-supplied user profiles that do not represent users.

5.2.3. Subject Deletion

As described in section 3.2.3.4.4 “Process Termination” on page 3-66, a subject is deleted when the MI instruction Terminate Process (TERMPR) is invoked.

Users using the system interactively can cause the subject operating on their behalf to be deleted by logging off the system. The subject associated with a batch job is terminated when it completes execution.

5.2.4. Changing Subject Security Attributes

A subject can augment its security attributes by calling a program that adopts authority, as described in section 3.2.3.5.2 “Program Adoption” on page 3-70 and section 6.2 “Discretionary Access Control” on page 6-4.

A subject can use CL commands to change some of the security attributes associated with its user profile, and the security administrator can change any subject’s security attributes using CL commands.

5.2.5. Trusted Subjects

A *trusted subject* is any subject that satisfies at least one of the conditions identified below:

- A subject that has the ability to violate the security policy and is “trusted” to do so in a controlled manner
- A subject that is relied on (i.e., “trusted”) to enforce the security policy

In AS/400, any subject that is executing with any of the special authorities described in section 5.2.6 “User Classes and Special Authorities” on page 5-4 is a trusted subject since it has the ability to violate at least part of the security policy. For example, a subject with the *ALLOBJ special authority can access any object.

In addition, AS/400 uses subjects with special authorities to enforce certain aspects of the security policy. For example, the QSPLMAINT system job (see section 4.1.3.4.5 “Spool Maintenance System Job” on page 4-47) is responsible for clearing the contents of the physical files used by the Spool component before the files are freed for use by another user. Since this job runs under the user profile QSYS, which has the *ALLOBJ special authority, it can access the database files even though they are owned by QSPL and have public authority set to *EXCLUDE.

5.2.6. User Classes and Special Authorities

When the *security administrator* creates a new user profile, the new user is assigned a particular *user class* depending on the task assigned to the new user. The user class attribute in the user profile is used to specify the user class. The user class controls the menu options that are available to the user.

AS/400 security has five hierarchical user classes:

- Security Officer (*SECOFR), who performs all security functions including creating security administrators
- Security Administrator (*SECADM)[3], who performs security-related functions such as creating user profiles
- System Programmer (*PGMR), who performs system programming functions
- System Operator (*SYSOPR), who also performs system maintenance and operation functions. The System Operator can back up the system and save and restore objects
- End User (*USER), who performs application functions

In addition to the user classes, the user's role is determined by the special authorities. Special authorities are used to specify the type of actions a user can perform on system resources. A user can be given zero or more special authorities. Special authorities are implemented as a 32-bit mask in the user profile. Special authorities are specified in the special authority (SPCAUT) field of the user profile. If the value of SPCAUT is user class (*USRCLS), then special authorities are granted to the user based on the user class. If *USRCLS is specified, specific special authorities cannot be specified for this user. The mapping of special authorities to user classes is shown in Table 5-1 on page 5-4.

Table 5-1. User CLasses and their Special Authorities

	USER CLASS				
Special Authority	***SECOFR**	***SECADM**	***PGMR**	***SYSOPR**	***USER**
*ALLOBJ	X	-	-	-	-
*SECADM	X	X	-	-	-
*JOBCTL	X	-	-	X	-
*SPLCTL	X	-	-	-	-
*SAVSYS	X	-	-	X	-
*AUDIT	X	-	-	-	-
*SERVICE	X	-	-	-	-
*IOSYSCFG	X	-	-	-	-

[3] The *SECADM user class is not the same as the *SECADM special authority

Special authorities also determine what special privileged MI instructions a user can issue. When the security administrator updates the special authorities field in a user profile, the corresponding special privileged instruction bits in the user profile are also updated.

- *ALLOBJ: All Object - allows the user to have *ALL access (see section 6.2.1 “Authorities” on page 6-4) to each object whether or not private authorities exist for the user. Even if the user has *EXCLUDE authority to an object, *ALLOBJ special authority still allows the user to access the object.
- *SECADM: Security Administration - allows the user to create, change, and delete user profiles. The security administrator can grant only those special authorities (excluding *SECADM) that the administrator has. A user, on the other hand, must have *SECADM and *ALLOBJ special authorities to grant the *SECADM special authority.
- *SAVSYS: Save System - allows the user to save and restore all objects on the system.
- *JOBCTL: Job Control - allows the user to change, display, hold, release, cancel, and clear jobs that are executing on the system or are on a job queue that has the operational control attribute set to YES (i.e., OPRCTL(*YES)). *JOBCTL also allows the user to display or copy files on output queues specified as (OPRCTL(*YES)). *JOBCTL special authority allows the user to IPL the system and control the subsystems.
- *SERVICE: Service - allows the user to perform service operations.
- *SPLCTL: Spool Control - allows the user to delete, display, hold, and release spool files that are owned by other users. *SPLCTL also allows the user to cancel, display, hold, and release jobs on job queues that are specified as OPRCTL(*NO).
- *AUDIT: Audit - allows the user to specify or change the audit system values to determine system-wide audit criteria. *AUDIT also allows the user to set auditing levels for individual objects and users.
- *IOSYSCFG: Input/Output (I/O) System Configuration - allows the user to change system I/O configurations.

5.3. Objects

Objects are passive entities that contain information and that have an interface available to untrusted subjects. The objects in AS/400 include both named objects and unnamed objects. *Named objects* are nameable entities that allow information to flow between subjects of different identities.

AS/400 is an object-based system and, as such, provides many different kinds of objects. This section examines the kinds of AS/400 objects provided by AS/400 and identifies which of these are TCSEC named objects. It describes the methodology used by the team to determine whether or not each AS/400 object was a named object, and provides a rationale for why or why not each is a named object. It also describes the security-related attributes of AS/400 objects and the access policy enforced on AS/400 objects. The object reuse policy is discussed in section 6.4 “Object Reuse” on page 6-17, and the audit policy related to objects is discussed in section 6.5 “Audit” on page 6-18.

5.3.1. Object Determination Approach

The initial list of AS/400 objects was supplied by IBM. Each kind of AS/400 object identified in the list was examined by the team to determine whether the AS/400 object could be categorized as a named object. Significant characteristics in making this determination included the following:

- The existence of interfaces by which the AS/400 object can be directly accessed by untrusted users.
- The potential of using the AS/400 object to support direct information flow between untrusted subjects of different identities (i.e., is the object shared?).

In addition, the team examined the list of AS/400 objects to determine if any were *public objects*. Public objects are objects that are readable by any user of the system, but modifiable only by trusted users.

The AS/400 objects are divided into the following three types:

MI Objects: As described in section 3.2.1.5 “MI Objects” on page 3-26, AS/400 provides MI objects at the SLIC interface. The object study of the MI objects is described in section 5.3.1.1 “MI Objects” on page 5-6.

Internal Objects: As described in section 4.1.1.2 “Internal Objects” on page 4-12, OS/400 uses internal objects to manage OS/400 data. Since there are no interfaces to access internal objects, the team eliminated internal objects from further consideration.

External Objects: As described in section 4.1.1.1 “External Objects” on page 4-2, AS/400 provides external objects at the OS/400 interface. The object study of the external objects is described in section 5.3.1.2 “External Objects” on page 5-7.

5.3.1.1. MI Objects

As discussed previously, MI objects are created and manipulated via MI instructions. OS/400 uses these MI instructions to create MI objects, which it uses as the building blocks for external objects. Users then use CL commands and APIs to access the external object, and OS/400 uses MI instructions to access the underlying MI objects on behalf of the user.

The AS/400 TCB uses two mechanisms to prevent users from accessing MI objects directly using MI instructions. These are described below.

Blocked MI Instructions: Non-TCB code can use only unblocked MI instructions to create and manipulate MI objects. Consequently, any MI objects that can be created and manipulated only by blocked MI instructions are not named objects since there is no TCB interface to directly access the MI object.

There are three *unblocked* MI instructions that create MI objects. Two of these create a space MI object and one creates an index MI object. When any of these MI instructions are invoked from User State, they create the object in User Domain and only in the QTEMP library. Since placing an object in the QTEMP library makes it non-sharable, the team concluded that neither the space MI object nor the index MI object was a named object when created by untrusted users.

Object Domain: If an attempt is made to access a System Domain MI object from User State using an unblocked MI instruction, a domain exception occurs. Therefore, the team concluded that all MI objects created in System Domain are not named objects.

AS/400 provides one sharable User Domain MI object. The *PGM OS/400 object is created from a User Domain MI program object. Since the MI object is directly accessible via unblocked MI instructions, the team concluded that the MI program object is a named object when it is in User Domain.[4]

The DAC policy for MI program objects is described in section 5.3.3 “DAC Policy” on page 5-9.

Some instances of MI objects are *public objects*. AS/400 provides one public object:

Entry Point Table: The EPT is an MI Space object that is in User Domain with read-only hardware storage protection so it cannot be changed from User State. It is described in more detail in section 4.1.1.2 “Internal Objects” on page 4-12.

5.3.1.2. External Objects

At the OS/400 interface, AS/400 provides external objects, which are built from MI objects. As described in section 4.1.1.1 “External Objects” on page 4-2, there are 47 types of external objects.

The team divided the external object types into the following three categories:

Non-shared Objects: Non-shared objects are objects whose access is restricted by AS/400 to only the object creator.

In AS/400, the QTEMP library (object of type *LIB) is a non-shared object. Addressability to each user’s QTEMP library is contained in the user’s Process Control Space (PCS). It is not placed in the machine context. Consequently, there is no TCB interface by which a user can obtain a system pointer to another user’s QTEMP library or to any objects in that library (since addressability to a library is required to get a system pointer to the objects in the library). Therefore, any time an object is placed in QTEMP, the instance of that object type is also non-shared.

There is also no way for a user to generate a system pointer to QTEMP or any objects in QTEMP since attempts to modify an existing system pointer so that it points to another object invalidate that pointer. There is no way for a user to change a space pointer so that it points to another segment because the MI instructions that manipulate space pointers perform bounds checking.

Finally, there is no way to pass a system pointer to another process because the Message Handling Component checks for embedded pointers and invalidates them by clearing the tag bits.

Objects Whose Access Is Restricted: In the evaluated configuration, there are instances of 12 types of external objects that are created during installation or IPL. Nine of these object types are related to networking, which is not being supported in the C2 configuration. Two are related to the Office Component, which is not under evaluation. One is for registering user exits, which are not needed in the TCB. The evaluated configuration prevents the use of TCB interfaces that manipulate these object types by setting the PUBLIC authority on the CL commands and APIs to *EXCLUDE. Therefore, the manipulation of these object types is restricted to trusted users, who are warned in the TFM against their use.[5]

[4] The MI program objects associated with IBM-supplied *PGMs are in System domain. Consequently, these MI program objects are not named objects.

[5] The team will test to ensure that DAC prevents the use of these interfaces by untrusted users.

These object types are listed in Table 5-2 on page 5-8.

Table 5-2. Excluded External Object Types

Object Name	Object Type	Description
Alert Table	*ALRTBL	Used for networking
Connection List	*CNNL	Used for networking
Class of Service Description	*COSD	Used for networking
Line Description	*LIND	Used for networking
Mode Description	*MODD	Used for networking
Document	*DOC	Managed by Office Component
Folder	*FLR	Managed by Office Component
Filter	*FTR	Used for networking
Exit Registration	*EXITRG	Used by US Component
Network Table Description	*NTBD	Used for networking
IPX Description	*IPXD	Used for networking
SQL Package	*SQLPKG	Used for networking

TCSEC Named Objects: The remaining external object types comprise the TCSEC Named Objects.

The named objects supported by AS/400 are shown in Table 5-4 on page 5-12.

5.3.2. Security Attributes of Objects

Several security attributes are associated with each external object. Each external object is made up of one or more MI objects, and each MI object is made up of one or more segments. Therefore, security attributes for external objects are stored within MI object headers and within segment headers. Additionally, security attributes may be stored within system pointers to the object and within various user profiles. Finally, if the object is secured with an authorization list, that authorization list may contain security attributes for the object. The security attributes for an object and their various locations are described below:

Authorized Pointer: A system pointer with a set of authorities stored within it. Addressability of an object is through a system pointer. A subject may be granted access to the object pointed to by the system pointer based on the authorities stored in that system pointer.

Segment Header: The Segment Headers contain information about the segments associated with the object. These include the following:

- SID Extender
- Domain - USER or SYSTEM
- Hardware Protection Class

Encapsulated Program Architecture (EPA) Header: As described in section 3.2.1.1 “MI Object Format” on page 3-17, each type of MI object has associated with it a data structure called the *EPA header*, which contains several security attributes of the MI object, as described below:

- Object type
- Owning user profile
- Owner’s authority
- Primary group profile
- Primary group authority
- Public authority
- Audit flag (see section 6.5.4.3 “Auditing of Objects” on page 6-24)
- No private authority flag (see section 6.2 “Discretionary Access Control” on page 6-4)
- No private less than public flag (see section 6.2 “Discretionary Access Control” on page 6-4)
- Pointer to the Authorization List that secures object (if one exists)

When an external object is made up of multiple MI objects (each of which contains an EPA header), it is the responsibility of the OS/400 external object manager to determine which EPA header is used in making DAC decisions at the OS/400 level and to set the values in each EPA header appropriately so that any DAC checks performed by SLIC pass.

Owning User Profile: The user profile of the user who owns an object contains the following security attributes related to the object:

Pointer to the User Profile of each user with private authorities to the object

Authorized User Profile: The user profile of each user who has private authorities to an object contains the following security attributes related to the object:

Authorized User’s Private Authority to the object

Authorization List: When an object is secured by an authorization list, the authorization list may contain security attribute information for the object. A user’s authority (either owner’s authority or his or her private authority) to an authorization list is used as his or her authority to each object secured by the authorization list.

Pointer to each object secured by that authorization list.

Public authority to authorization list (When public authority of an object secured by authorization list is *AUTL, then public authority to the authorization list object is used as the object’s public authority.)

5.3.3. DAC Policy

AS/400 enforces a DAC policy on all named objects as described in this section. The DAC mechanisms that enforce the DAC policy are described in section 6.2 “Discretionary Access Control” on page 6-4.

Addressability to the byte-addressable portion (i.e., associated space) of any object is through a space pointer. A subject possessing a space pointer to an object is granted both read and write access to the associated space of that object (within the constraints of the hardware storage protection).

Access to non-shared objects is limited to the object creator. Enforcement of this policy is described in section 5.3.1.2 “External Objects” on page 5-7.

Unlike non-shared objects, named objects can be shared among subjects. The use of authorities in determining whether a subject is granted access to a named object are discussed in section 6.2 “Discretionary Access Control” on page 6-4. In general, the policy enforced is summarized in Table 5-3 on page 5-10. There are numerous exceptions and caveats to the general DAC policy statement, and for many object types, the policy enforced is more restrictive than the general policy. Column 4 of Table 5-4 on page 5-12 either refers to the section of this report that contains a more detailed discussion of the policy for that particular object type or it describes the policy, if it is less restrictive.

Table 5-3. Operations and Associated Authorities on Objects

Operation	Object Authority	Data Authority	Notes
Create Object	Not Applicable	Not Applicable	*ADD and *READ to library; *ADD to owning user profile
Create Object when REPLACE(*YES) is specified	*OBJMGT and *OBJEXIST	*READ	*ADD and *READ to library; *ADD to owning user profile
Delete Object	*OBJEXIST		*EXECUTE to library
Read Attributes	*OBJOPR		
Write Attributes	*OBJMGT		
Read Data	*OBJOPR	*READ	
Add New Data	*OBJOPR	*ADD	
Modify Data	*OBJOPR	*UPD	
Delete Data	*OBJOPR	*DLT	
Execute Program	*OBJOPR	*EXECUTE	*EXECUTE to library

Table 5-3. Operations and Associated Authorities on Objects

Operation	Object Authority	Data Authority	Notes
Copy (when to-file is physical file)			
-Object to be created	*OBJOPR	*READ	
-To-file if CRTFILE (*YES) is specified			*ADD to library
-To-file, if it exists and new member to be added	*OBJMGT and *OBJEXIST	*ADD	
-To-file, if file and member exist, and *ADD option specified	*OBJOPR	*ADD	
-To-file, if file and member exist, and *REPLACE option on	*OBJMGT and *OBJOPR	*ADD and *DLT	
Accessing directory objects			
-Directories in path prefix		*X	
-Directory when pattern is specified (* or ?)		*R	

The information contained in each column of Table 5-4 on page 5-12 is discussed below:

- Column 1: Object name (e.g., Library)
- Column 2: Object type (e.g., *LIB)
- Column 3: Domain (System/User) of each object
- Column 4: Identifies whether the hardware write protection is used to protect the object when it exists in User Domain. Also describes any unique DAC policy that applies to that external object type

Table 5-4. OS/400 Named Objects

Object Name	Object Type	Domain	Description/Policy
Authorization List	*AUTL	System	Owner cannot control who reads it or who uses it. See section 6.2.2 “Authorization Lists” on page 6-6.
Block Special File	*BLKSF	System	
Binding Directory	*BNDDIR	System	
C Locale Description	*CLD	System	
Class	*CLS	System	
CL Command	*CMD	System	
Change Request Description	*CRQD	System	
Controller Description	*CTLD	System	
Device Description	*DEVD	System	
Directory	*DIR	System	
Data Area	*DTAARA	System	
Data Dictionary	*DTADCT	System	
Data Queue	*DTAQ	System	
Edit Description	*EDTD	System	
Forms Control Table	*FCT	System	
Logical File	*FILE	System	DAC policy described in section 6.2 “Discretionary Access Control” on page 6-4.
Physical File	*FILE	System	DAC policy described in section 6.2 “Discretionary Access Control” on page 6-4.
Device File	*FILE	System	Device files include printer files, tape files, and display files. DAC policy described in section 4.1.1.1.2 “Data Management” on page 4-9.
Font Resource	*FNTRSC	System	
Font Table	*FNTTBL	System	

Table 5-4. OS/400 Named Objects

Object Name	Object Type	Domain	Description/Policy
Form Definition	*FORMDF	System	
Graphics Symbol Set	*GSS	System	
Job Description	*JOBD	System	
Job Queue	*JOBQ	System	DAC policy described in section 4.1.3.4.3.5 “Job Queues” on page 4-39.
Job Scheduler	*JOBSCD	System	
Journal	*JRN	System	
Journal Receiver	*JRNRCV	System	
Library	*LIB	System	
Locale	*LOCALE	System	
Menu	*MENU	System	
Module	*MODULE	System	
Message File	*MSGF	System	
Message Queue	*MSGQ	System	
Output Queue	*OUTQ	System	DAC policy described in section 4.1.3.4.4.3 “Output Queues” on page 4-42.
Overlay	*OVL	System	
Page Definition	*PAGDFN	System	
Page Segment	*PAGSEQ	System	
Print Descriptor Group	*PDG	System	
Program	*PGM (IBM Supplied)	System	To run a program requires *EXECUTE data authority.

Table 5-4. OS/400 Named Objects

Object Name	Object Type	Domain	Description/Policy
Program	*PGM (User created)	User	Associated Space is hardware write-protected. To run a program • via the CL command CALLPGM, requires *OBJOPR object authority and *READ and *EXECUTE data authorities • as the CPP of a CL command, only requires *EXECUTE data authority • via MI interfaces requires *EXECUTE to underlying MI program object.
Panel Group	*PNLGRP	System	
Product Definition	*PRDDFN	System	
Product Load	*PRDLOD	System	
PSF Configuration	*PSFCFG	System	
Query Management Form	*QMFORM	System	
Query Management Query	*QMQRY	System	
Query Definition	*QRYDFN	System	
Reference Code Translation Table	*RCT	System	
Save File	*SAVF	System	
Subsystem Description	*SBSD	System	
Information Search Index	*SCHIDX	System	
Spelling Aid Dictionary	*SPADCT	System	
Service Program	*SRVPGM (IBM supplied)	System	
Service Program	*SRVPGM	User	Associated Space is hardware write-protected.

Table 5-4. OS/400 Named Objects

Object Name	Object Type	Domain	Description/Policy
Stream File	*STMF	System	
Symbolic Link	*SYMLNK	System	
Table	*TBL	System	
User Index	*USRIDX	System	Domain set to User when instance of object placed in user's QTEMP library. Otherwise, Domain set to System and follows general policy.
User Profile	*USRPRF	System	DAC policy for User Profile objects and for user profile objects when they are used as group profiles are described in section 5.3.3.1 "User Profile and Group Profile Objects" on page 5-15.
User Queue	*USRQ	System	Domain set to User when instance of object placed in user's QTEMP library. Otherwise, Domain set to System and follows general policy
User Space	*USRSPC	System	Domain set to User when instance of object placed in user's QTEMP library. Otherwise, Domain set to System and follows general policy
Validation List	*VLDL	System	
Workstation User Customization	*WSCST	System	

5.3.3.1. User Profile and Group Profile Objects

There are two types of profiles on the AS/400 system, user profiles and group profiles. Both types of profiles are created from the *USRPRF external object.

5.3.3.1.1. User Profiles

A user is defined by a user profile object.

The following attributes are associated with user profiles:

- User profile name: A name that identifies the user to the system.
- Password: A one-way encrypted character string used to authenticate the user.
- Expired Password Indicator: Indicates that a user's password has been set to expired and that it must be changed the next time the user signs on.
- Status: User profile status (i.e., enabled or disabled).
- Previous sign-on date: Contains the date the user previously signed on to the system.

- Sign-on attempts not valid: The number of sign-on attempts that were not valid since the previous successful sign-on.
- Current library: The default library.
- Initial menu: Initial menu (*MENU object) displayed when the user signs on.
- Initial menu library: Library containing user's initial menu.
- Initial program: Program (*PGM) called when the user signs on.
- Initial program library: Library containing user's initial program.
- Limit capabilities: Limit the user's ability to enter commands, initial menu, and current library.
- Text: The text description of the object.
- Display sign-on information: Controls whether sign-on information display is shown when the user signs on.
- Limit device sessions: Controls whether a particular user can be signed on more than one display device at a time.
- Keyboard buffering: Controls whether keyboard buffering is in effect when the user signs on.
- Date password last changed: Contains date the password was last changed.
- Password expiration interval: Contains length of time that may elapse before the user is required to change his or her password.
- Date password expires: Contains the date the password expires.
- User class: Specifies user's class, as described in section 5.2.6 "User Classes and Special Authorities" on page 5-4.
- Special authorities: Specifies user's special authorities, as described in section 5.2.6 "User Classes and Special Authorities" on page 5-4.
- Group profile: Pointer to group profile that describes the group to which this user belongs.
- Owner: Specifies whether the user profile or the group profile is the owner of any objects created by this user.
- Group Authority: Specifies the private authority given to the group profile for any objects created by this user.
- Group Authority type: Specifies the group authority type given to the profile for any objects created by this user. The authority given can be a private authority or a primary group authority.
- Supplemental Group profiles: 0 - 15 group profiles to which this user belongs.
- Assistance level: The level of interface that is used after the user signs on the system.
- Accounting code: Accounting code associated with this user.
- Message queue: Specifies message queue (*MSGQ object) for messages.
- Message Queue Library: Library containing message queue.
- Message queue delivery: Used to indicate whether the user is interrupted when a new message arrives on the queue.

- Message queue severity: Used to determine the lowest-level messages that are delivered to the user.
- Job description: The job description (*JOBD) that is used when the user submits jobs.
- Job description library: The library containing the job description.
- Maximum storage allowed: The maximum amount of auxiliary storage that may be used for objects owned by this profile.
- Storage used: The amount of auxiliary storage already used.
- Highest scheduling priority: The highest scheduling priority that is allowed for any job submitted by this user.
- Output queue: Specifies default output queue (*OUTQ).
- Output Queue Library: Library containing default output queue.
- Printer device: Specifies default printer device.
- Special environment: Used to indicate the environment the user operates in after signing on (e.g., AS/400, System/38, System/36).
- Attention program: Specifies the Attention-Key-Handling-Program (ATNPGM) (*PGM) activated when the attention key is pressed.
- ATNPGM Library: Library containing the ATNPGM.
- Authority: Specifies the public authority to the user profile.
- OBJAUD: The object auditing value for a user profile works with the object auditing value for an object to determine whether the user's access of an object is audited, as described in section 6.5.4.3 "Auditing of Objects" on page 6-24.
- AUDLVL: Specifies security-relevant actions that are recorded in the audit journal for this user, as described in section 6.5.4.2 "Auditing of Actions" on page 6-23.
- GID: Group identification number for group user profiles.
- UID: User identification number.
- Home Directory: Home directory for this user.
- Sort sequence: A table that controls the order in which graphic characters are arranged during sort operations or other comparisons.
- Sort sequence library: The library that contains the sort sequence table.
- Language identifier: The language identifier used for this user.
- Coded character set identifier: The coded character set identifier (CCSID) used for this user.
- Locale job attributes: Job attributes that are taken from the locale specified for this user when the job is initiated.
- Locale: The path name of the locale that is assigned to the LANG environment variable for this user.
- User options: Personalization options that are used to control the direction in which the roll keys move text lines on the display and the default level of information that is shown to the user for prompting of CL commands and on certain system displays.

- Document password: The password that allows Document Interchange Architecture (DIA) users to protect personal distributions from being used by people working on their behalf.

In addition to the attributes listed above, user profile objects contain other information as described below:

- Pointer to each object owned by the user profile
- Pointer to each object to which the user has private authority and the associated private authorities
- Pointer to each object to which the user has primary group authority
- For each owned object, a pointer to each user profile that has a private authority to the object and the associated private authority
- Special privileged instructions mask: A 4-byte field indicating the set of special privileged instructions that the user can execute, as described in section 3.2.2.2.4 “Special Privileged Instructions” on page 3-41. There are no TCB interfaces to directly update this field. Rather, the field is updated as a result of changing the special authorities associated with a user profile.

Finally, since user profiles are objects, they also contain information in their EPA header, as described in section 5.3.2 “Security Attributes of Objects” on page 5-8.

The interfaces to manipulate the user profile object provide field-level protection in that they require special authorities to update certain fields, as described in the following sections.

5.3.3.1.1.1. Creating a User Profile

*SECADM special authority is required to create a user profile. When a user profile is created, the creating user is made the owner and is given *ALL authority to the user profile. The user profile object is given private authorities to itself that are sufficient to allow the user represented by the user profile to update certain fields in the user profile object, as described in section 5.3.3.1.1.2 “Changing a User Profile” on page 5-18. This set of private authorities is the following: *OBJMGT and *CHANGE.

In the TFM, the administrator is warned to set the public authority for user profiles to *EXCLUDE to prevent misuse of user profiles.

5.3.3.1.1.2. Changing a User Profile

Users with no special authorities can change the following fields in their own user profile. Users must have *OBJMGT and *USE to their user profile to change these fields, and they must have *USE authority to the objects for fields they are changing:

- Initial program
- Initial menu
- Job description
- Message queue
- Output queue
- Attention-key-handling program
- Current library
- Assistance level
- Text
- Keyboard buffering
- Document password
- Message queue delivery
- Severity code filter
- Print device

- Sort sequence
- Language identifier
- Country identifier
- Coded character set identifier
- Locale job attributes
- Locale
- User options
- Home directory

Users with *SECADM special authority can change all of the fields in any user profile for which they have *OBJMGT and *USE authority except the audit fields. Administrator must have *USE authority to the objects for fields they change.

A user with *AUDIT special authority can change the OBJAUD and AUDLVL auditing values of any user profile regardless of any private or public authorities on that user profile.

5.3.3.1.1.3. Deleting a User Profile

Users with *SECADM special authority can delete a user profile if they have both *OBJEXIST and *USE authority to the user profile. The message queue associated with the user profile is deleted if it is owned by that user profile and if the user also has *OBJEXIST, *USE, and *DLT to that message queue.

A user profile that owns objects cannot be deleted, a group profile that has members cannot be deleted, and a group profile that is the primary group of any object cannot be deleted.

5.3.3.1.2. Group Profiles

A group profile is a user profile object that is used as a method for grouping together users who share the same resources. Authorities to objects can be defined in the group profile, and all user profiles in the group can access these objects based on the authorities associated with the group profile.

The create user profile CL command is used to create both user profiles and group profiles. A user profile is recognized as a group profile only when it is first specified as the group profile for a user.

The TFM warns administrators to assign group profiles a password of *NONE to prevent them from being used for signing on.

5.4. Devices

Devices on AS/400 are the peripherals that are used for input and output. They consist of direct access storage device (DASD), tape, optical, diskette, printer, and workstation. These devices are physically attached to the AS/400 Central Electronic Complex (CEC) through IOPs via standard interfaces, i.e., SCSI and Artic. See section 3.1.1.2 “Input/Output” on page 3-12 for a description of the hardware, and Appendix A, “Evaluated Hardware”, on page A-1 for a list of the evaluated devices.

From a user perspective, the AS/400 devices are represented by a named, external object called a Device Description (*DEVD). See section 4.1.1.1 “External Objects” on page 4-2. A unique device description exists for each I/O device on the system, except DASD. The device description defines the attributes of the I/O device and must exist for a user to use the device. The DAC to the device is controlled through the device description object. Users must have *USE authority to the device description to use the device. Note that *USE does allow write operations to output devices.

The DASD on a single level store model is all part of the 64-bit address space and is not directly accessible by the end user. DASD is controlled by SLIC storage management. See section 3.2.3.3.2 “Virtual Storage and Single Level Store” on page 3-48 for details.

Section 6
TCB Protection Policies

This section describes the various features that are used to protect the resources from unauthorized access by the subjects on the system. The features discussed include identification and authentication, discretionary access control, object reuse, and auditing.

6.1. Identification and Authentication

AS/400 requires all users to identify and authenticate themselves before they are allowed to access system resources. Users are identified by a user profile and authenticated by a password. Each user has a unique user profile. A password of one to ten characters in length is used to authenticate users. (The Trusted Facilities manual [TFM] recommends a minimum password length of five characters.) This section describes the identification and authentication mechanism on AS/400.

6.1.1. Sign-on Process

The work management component of the AS/400 presents the user with the sign-on screen. The AS/400 sign-on screen is processed through a system job called the interactive subsystem monitor job (see section 4.1.3.3 “Job Management” on page 4-23).

The interactive subsystem monitor job is started automatically by the system during the IPL. The interactive subsystem monitor job controls the displaying of the sign-on screen and processes the input from the sign-on screen.

When the user enters a user profile name and a password on the sign-on screen, the following events take place:

- The user profile name is validated to ensure that the string entered is one of the allowable character sets for a user profile name. If the string contains an invalid character, the user receives an error message indicating that the user profile name is invalid.
- The user profile name is used to locate the user profile object on the AS/400. If the user profile object is not found, then the user receives an error message indicating that the user profile does not exist.
- The password for the user profile is validated to ensure that it is the password associated with the user profile. If the password is not valid, then the user receives an error message indicating that the password is incorrect.

- The user profile, if found, is checked to ensure that it has not been disabled. If the user profile is disabled, then the user receives a message on the sign-on screen indicating that the user profile cannot sign on to the system.
- The authority of the user profile to the objects that are used to start the interactive job are checked. If the user profile does not have access to these objects, then the user receives a message indicating that the user profile has insufficient access to the objects and the sign-on process fails.

The following subsection discusses interactive job initiation and the identification and authentication process in more detail.

6.1.1.1. Interactive Job Initiation Flow, Interactive Subsystem Monitor

As part of the identification and authentication process, the following programs are involved. These programs run in the interactive subsystem monitor job prior to starting the job for the user who is signing on to the system (see section 4.1.3.3 “Job Management” on page 4-23).

- Subsystem monitor mainline program. This program has two functions:
  - Calls the interactive job initiation event handler program to process interactive sign on requests
  - Initiates the interactive user job
- Interactive job initiation event handler. This program performs three functions:
  - Calls the retrieve user profile information program to retrieve the values of the user profile for the user who is signing on
  - Calls the user profile verification program to verify the user profile and password. The user profile verification program returns the result of the comparison to the interactive job initiation event handler
  - Calls the interactive job sign-on processing program to increment the count of unsuccessful sign-on attempts for a user profile if an incorrect password was entered
- User profile verification. This program performs the following functions:
  - Verifies the user profile’s name and password
  - Certifies the disabled user profile attribute
  - Performs authority checks for the objects that are used to start the job (e.g., display station, current library, etc.)
  - Audits invalid user profile names and invalid passwords
- Retrieve user profile information. This program performs the following:
  - Retrieves information about the user that is signing on the AS/400 system.
- Interactive job sign on processing. This program does the following:
  - Increments the count of unsuccessful sign-on attempts

  - Performs the password expiration checks. If the password for the user profile has expired, then the user must change the password or sign off the system.
  - Displays the sign-on information for the user who is signing on the system. The sign-on information includes the date of the last sign-on, the date the password was last changed, and the number of days before the password expires.

6.1.2. Password Verification and Management

A password of one to ten characters is used to authenticate users. The password is initially set by the security administrator when the user profile was created. Users may change their passwords through a CL command or through an API. The password can be set to *NONE, which prevents the use of the user profile for interactive sign on.[1]

DES encryption is used to encrypt the user profile name using the user profile password as a key. The result of this encryption is then compared to the encrypted password stored in the Authorized User Table. If the two values are the same, then the user is authenticated. If the values are different, the user is allowed to retry the password. The exact number of retries is determined by the security administrator. The AS/400 also can restrict where users are allowed to sign on (physically) by restricting authority to the display stations.

Identification and authentication information is maintained in an internal system domain object. The internal object is known as the Authorized User Table or AUT.

AS/400 gives the security administrator flexibility in password management. The security administrator can change system values (see section 4.1.3.13 “System Values” on page 4-65) to control the following:

- Password lifetime
- Password composition rules to eliminate trivial passwords (e.g., minimum length, maximum length, restricted characters, no repeated passwords, restrict adjacent digits, no repeated characters, digit required in password)
- Number of active sessions allowed for this user
- Number of days between password changes
- Retry limit (if a user fails to enter the password correctly)
- Action to disable the user’s profile when the limit is reached

The TFM provides guidelines on setting these system values.

6.1.3. Sign-on Failure

A user may fail to sign on the system due to several reasons (invalid user profile name, invalid password, not authorized to objects used to start the job, etc.). All failed login attempts on the AS/400 are auditable events. The number of times a user can attempt to sign on the system and the appropriate actions that the system can

[1] Previously submitted batch jobs still run.

take are determined by system values. There are a number of actions that can be taken when the limit is reached:

- Disable display station
- Disable profile (change the *Status* attribute in the user profile to *Disabled*)[2]
- Disable display station or profile

The date and time of all failed login attempts for a user are recorded in the security audit log.

6.2. Discretionary Access Control

The AS/400 system allows users to protect objects using *authorities* (permissions) and *authorization lists*. In addition, the system also requires a user to have special authorities (privileges) to access certain external objects by commands and APIs and some MI instructions (special privileged instructions). This section discusses authorities and authority checking. Section 5.2.6 “User Classes and Special Authorities” on page 5-4 discusses special authorities, and section 3.2.2.2.4 “Special Privileged Instructions” on page 3-41 discusses special MI instructions.

6.2.1. Authorities

Authority can be granted to individual users, groups of users, and all users (known as public - *PUBLIC). There are four types of authorities: owner, primary group, private, and public.

Owner authority refers to the authority of the owning user profile. All objects on the system have an associated owning user profile. When an object is created, the owning user profile defaults to the user profile of the creating process. The owner has all authorities (*ALL) to an object when created.

Primary group authority refers to the authority of the primary group for the object. An object may or may not have a primary group. Primary group authority is a special type of private authority for groups that is stored with the object, not with the group profile.

Private authority refers to an authority that is explicitly granted to a particular user profile. Private authorities to objects are stored in the user or group user profiles. When both user and group profiles have private authorities for the same object, the authority in the user profile takes priority over the authority in the group profile.

Public authority applies to all users on the system. The public authority is used in the absence of owner, primary group, or private authority.

There are two categories of authorities: object authorities and data authorities. Object authority defines the operations that can be performed on the object as a whole. Data authority defines the operations that can be performed on the contents of an object.

[2] A user profile can be enabled by a security administrator.

The following list specifies possible values for object authorities:

Object Operational (*OBJOPR) allows a user to look at the description of an object (type, text, etc.) and use the object as determined by the specified data authorities. To open a file, for example, the user must have this authority.

Object Management (*OBJMGT) allows the user the ability to specify authorities for the object, move and rename the object, and add members to database files.

Object Existence (*OBJEXIST) allows the user control of the object's ownership and the ability to delete the object.

Object Alter (*OBJALTER) allows the user to add, clear, initialize, and reorganize members of a database file.

Object Reference (*OBJREF) allows the user to specify a database file as the parent in a referential constraint.

Authorization List Management (*AUTLMGT) allows users to add, remove, and change users and their authorities on an authorization list. *ALLOBJ special authority or ownership of the authorization list is needed to give *AUTLMGT authority for the authorization list to another user. This authority pertains only to authorization lists.

The following list specifies possible values for data authorities:

Read (*READ) allows a user to display the contents of the object (e.g. displaying records of a file) and add spool files to an output queue.

Update (*UPD) allows a user to change entries (e.g., records) in the object.

Add (*ADD) allows a user to add entries to the objects.

Delete (*DLT) allows users to remove entries (e.g., records and messages) from the object.

Execute (*EXECUTE) allows users to run a program, service program, or SQL package, or search for an object in a library or directory.

Exclude (*EXCLUDE) prevents user access to the object. Exclude authority is different than having no authority. Based on the algorithm for granting authority, having no authority implies that the public authority defined for the object is used. *EXCLUDE is both a data and an object authority.

Authorities can be used individually or in predefined system supplied groups. The system supplied groups are *ALL, *CHANGE, *USE, and *EXCLUDE. Figure 6-1. “Authority Groupings” on page 6-6 lists the mapping for authority groupings

In addition, system supplied groups of *RWX, *RW, *RX, *R, *WX, *W, and *X are provided for use with the integrated file system. Figure 6-2. “Additional Authority Groupings” on page 6-7 lists the mapping of these authorities groupings.

Individual Authorities	**System Grouping**			
	***ALL**	***CHANGE**	***USE**	***EXCLUDE**
Object Authorities				
*OBJOPR	X	X	X	
*OBJMGT	X			
*OBJEXIST	X			
*AUTLMGT				
*OBJALTER	X			
*OBJREF	X			
Data Authorities				
*READ	X	X	X	
*ADD	X	X		
*UPD	X	X		
*DLT	X	X		
*EXECUTE	X	X	X	

Figure 6-1. Authority Groupings

6.2.2. Authorization Lists

Authorization lists are external objects of type *AUTL. They are used to assign particular authorities for different users to a set of objects. Users can have different authorities to a common set of objects secured by the authorization list. All objects except user profiles and other authorization lists can be secured by an authorization list. An object can be secured only by one authorization list at a time, but an authorization list can secure many objects. The implementation of authorization lists is illustrated in Figure 6-3. “Authorization List Implementation” on page 6-8.

Only the owner of the object, a user with all object (*ALLOBJ) special authority, or a user with all (*ALL) authority to the object can secure or unsecure an object with an authorization list. Everyone can read authorization lists, regardless of private authorities. Other accesses such as modification are controlled, but read is not.

The Authorization List Management (*AUTLMGT) object authority applies only to authorization lists. Users with *AUTLMGT authority are allowed to add and remove users on the list and change those users’ authorities within the following constraints:

- A user with *AUTLMGT authority can give only the same or less authority to others.
- A user with *AUTLMGT authority can remove a user from the list only if the *AUTLMGT user has a superset of authorities to the list than the user profile name being removed.

When an object is secured by an authorization list, additional information can be used from the list. The *PUBLIC authority of the authorization list, stored in the authorization list object header, has multiple uses. It is the public authority to the authorization list object itself and when the public authority of an object is set

Individual Authorities	**System Grouping**						
	***RWX**	***RW**	***RX**	***R**	***WX**	***W**	***X**
Object Authorities							
*OBJOPR	X	X	X	X	X	X	X
*OBJMGT							
*OBJEXIST							
*AUTLMGT							
*OBJALTER							
*OBJREF							
Data Authorities							
*READ	X	X	X	X			
*ADD	X	X			X	X	
*UPD	X	X			X	X	
*DLT	X	X			X	X	
*EXECUTE	X		X		X		X

Figure 6-2. Additional Authority Groupings

to *AUTL, then the public authority in the authorization list object header is also used as the object’s public authority. An authorization list must be empty before it can be destroyed.

6.2.3. Sources of Authority

As discussed in section 5.2 “Subjects” on page 5-2, authority information is stored in the following locations:

- Object header
- Owning user or group profile
- Primary group profile
- Authorized user or group profile
- Authorization list
- System pointer

6.2.4. Access Control Algorithm

The AS/400 system checks access to objects at the OS/400 level and at the SLIC level. Every MI instruction that manipulates objects via a system pointer (i.e., modifies, destroys, retrieves) performs an access check. Additional access checks are performed at the OS/400 level. Access checks made at the OS/400 level are reflected in specific CL commands and APIs.

Although there are several ways to invoke an access check, the algorithm that is used to verify authority is the same. This algorithm is primarily encoded in a SLIC authorization routine. Other authority macros and

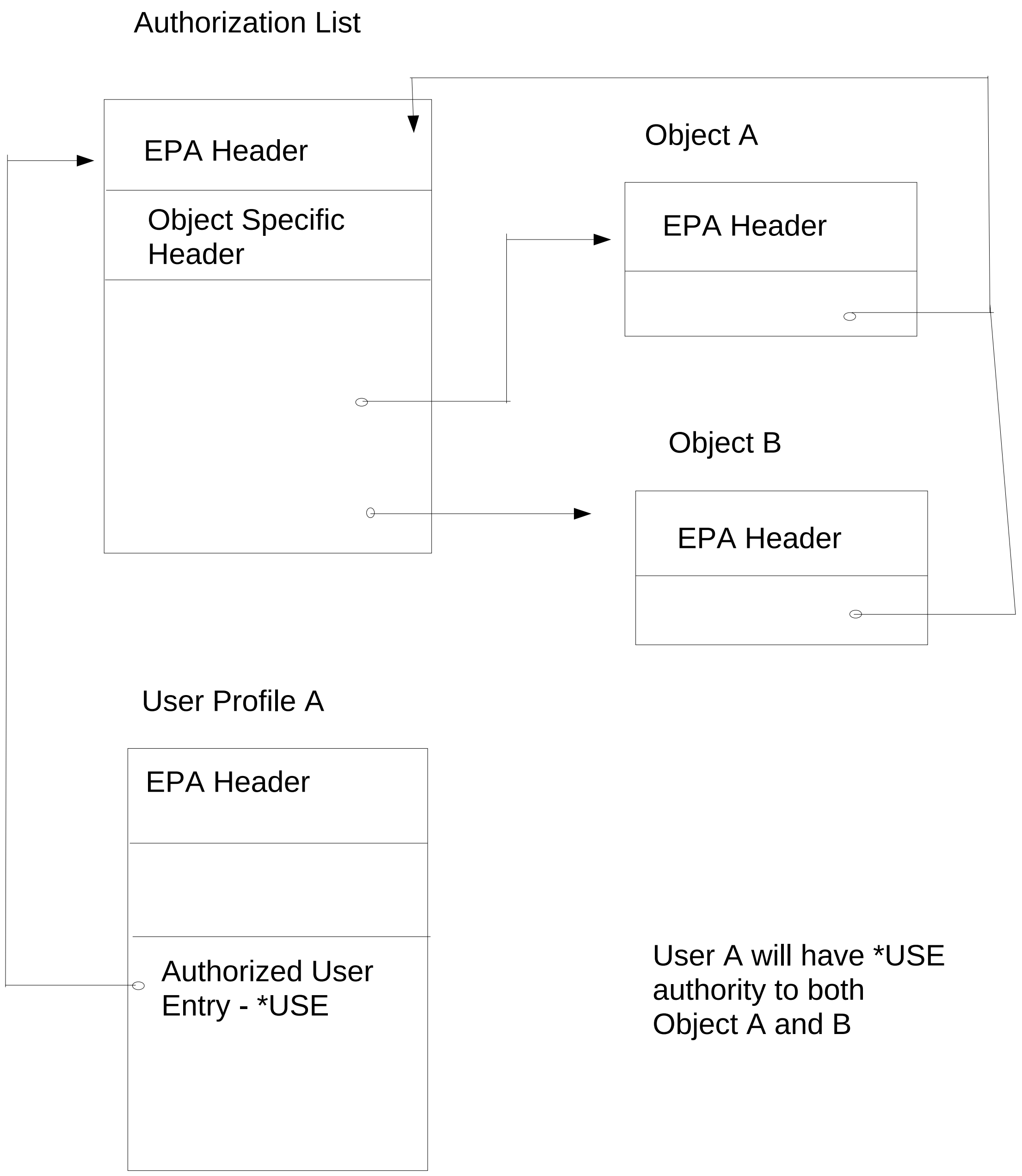


Figure 6-3. Authorization List Implementation

MI instructions ultimately call the SLIC authorization routine to perform authority checking. The authorization routine accumulates authority as it is located into a 16-bit mask. The caller of the authorization routine passes the required authority. This authority is compared against the authority associated with the process. If sufficient authority is located in the process, then the authority check passes; otherwise it fails.

Authority is stored in either the user profile (private), the object header (owner, primary group, and public) or authorized system pointers. The user profile contains 16 bits to indicate private authority. Each bit has a reserved meaning. To determine whether a user process has authority to an object, the authority checking algorithm compares the accumulated authority of the user process to the required authority of an object.

The system checks authority to an object in the following sequence:

1. Check System Pointer
2. Perform Quick Check
3. Process User Profile
4. Process Group Profile
5. Check Public Authority
6. Authority Sufficient Test
7. Check Program Adoption

When the address of an object is obtained, appropriate authority to the library is required in order to get a pointer to the object. First authority is required to the library to get addressability to an object, and then authority is required to the object itself. This requires two authority checks. For each check, authority is verified using the access control algorithm in Figure 6-4. “Authority Algorithm” on page 6-10. If the user process does not have the appropriate authority to the library, an exception occurs and addressability to the object is not available to the process.

The access control algorithm is illustrated in Figure 6-4. “Authority Algorithm” on page 6-10. Each step in the algorithm is explained below

.

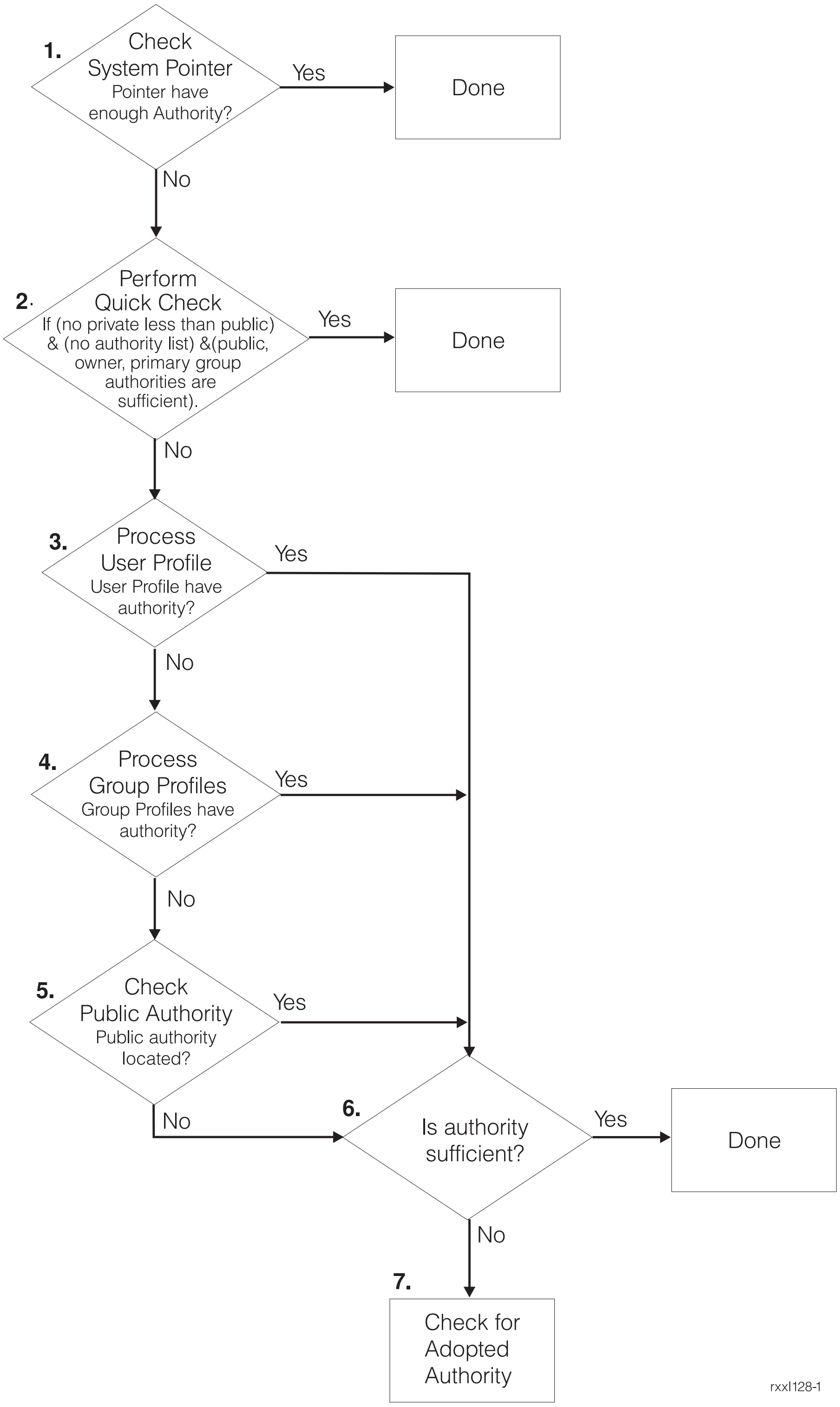


Figure 6-4. Authority Algorithm

- **Step 1 - Check System Pointer**

  The system pointer to the object may contain authority. If the authority in the pointer is a superset of the required authority, then access checking ends, indicating sufficient access. Otherwise, the authority in the pointer is accumulated in the authority mask and the algorithm continues to step 2.

- **Step 2 - Perform Quick Check**

  This step was added to the algorithm for better performance. This step checks for the following conditions:

  - No private authorities
  - No private authorities less than the public authority of the object
  - The object is not secured by an authorization list
  - The public authority for the object and the owner authority and the primary group authority, if present, are sufficient for access

  If the previous conditions are all true, then the algorithm terminates, indicating sufficient access. Otherwise, the algorithm checks for authority in the user profile.

- **Step 3 - Check User Profile**

  There are four sources of authority that can be checked as part of processing the user profile:

  - *ALLOBJ special authority located in the user profile
  - The pointer to owned object
  - Private authority to an object
  - User's authority on the authorization list

Several steps are involved in checking the authority located in the user profile. These steps are illustrated in Figure 6-5. “Process User Profile Steps” on page 6-12.

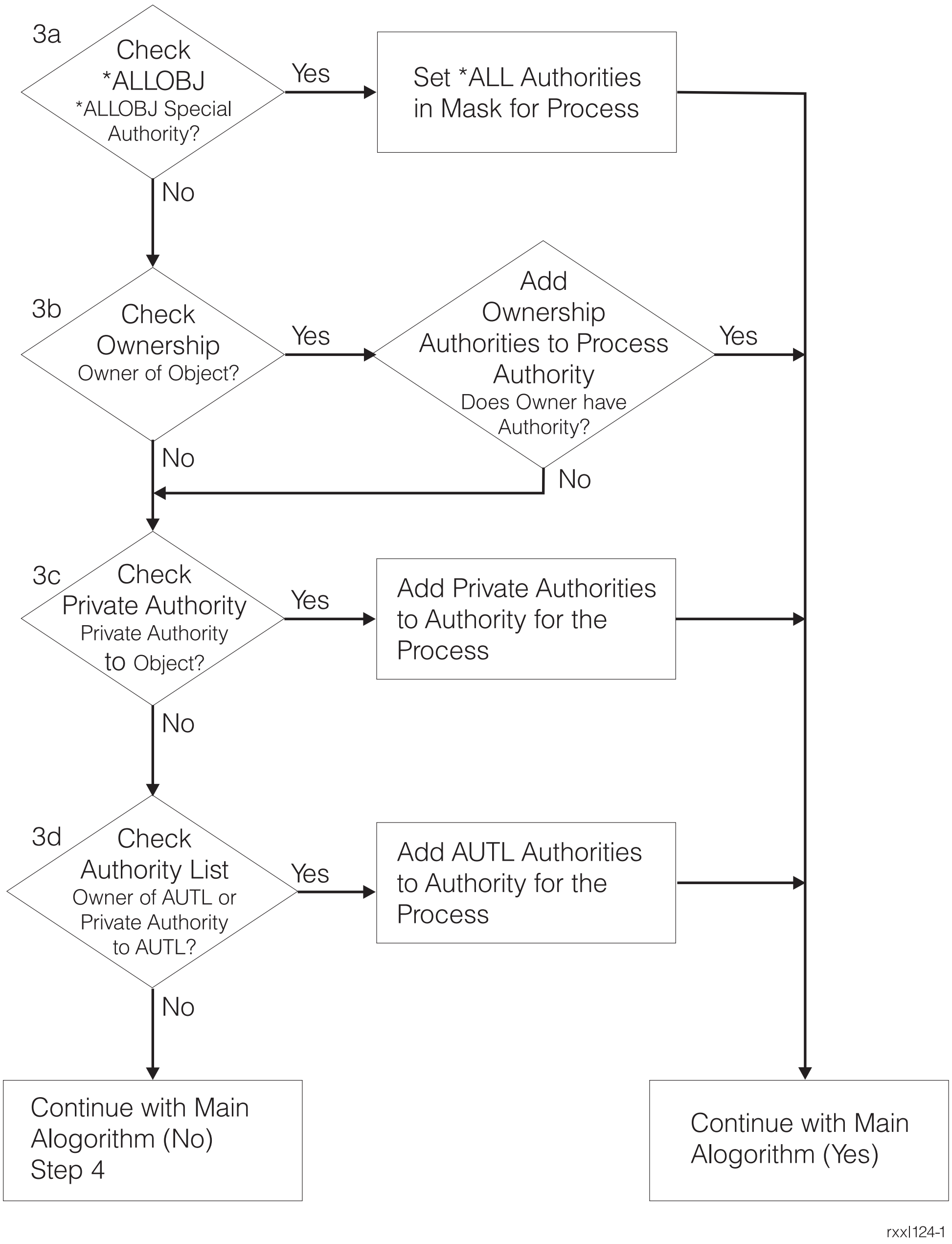


Figure 6-5. Process User Profile Steps

The first part of processing the user profile is to determine whether the profile contains the *ALLOBJ special authority. If the user profile has *ALLOBJ, then sufficient authority has been found and the access checking algorithm terminates.

If the user profile does not have *ALLOBJ, then the algorithm checks the owner authority. If the process user profile is the owner of the object, then the owner authority is determined. If this authority is sufficient, then access checking can terminate. If authority is located but is insufficient, then the algorithm skips to the check program adoption step. If no authority is located (i.e., user process is not object owner or owner has no authority), then the algorithm checks for private authority in the user profile.

If the user profile is in the primary group of the object, the primary group authority is determined. If this authority is sufficient, then access checking can terminate. If authority is located but is insufficient, then the algorithm skips to the check program adoption step. If no authority is located (i.e., user process is not the primary group or the primary group has no authority), then the algorithm checks for private authority in the user profile.

The access algorithm checks the no private authority flag in the object header to determine whether any private authorities exist. If there are no private authorities, then the algorithm advances to the check group user profiles step. Otherwise, the private authorities are further analyzed.

If private authority to the object is located but is insufficient, then the algorithm returns to the main algorithm, which skips to check the program adoption authorities. If no private authorities are located, then the object authorization list is checked. (An object authorization list contains public and private authorities for a group of objects.)

If the process user profile is the owner of the authorization list or has private authority to the authorization list, then authority is obtained from the authorization list and is added to the process authority.

The check user profile step of the access algorithm results in one of three cases:

a. Authority has been located and is sufficient.

b. Authority has been located but is insufficient.

c. Authority has not been located.

If authority has been located and is sufficient, the algorithm can terminate and access to the object is granted. If authority is located and is insufficient, the algorithm skips to the check program adoption step. If authority has not been located in processing the user profile, then the group profiles are processed for authorities.

- **Step 4 - Check Group Profiles**

A group profile is a special type of user profile object. Group profiles are processed in the same manner as user profiles. In this step, the profile that is searched for authorities is the group profile, rather than the user profile. Three sources of authorities for the group profiles are as follows:

a. Group’s *ALLOBJ special authority

b. Group’s authority to an object

c. Group’s authority on authorization lists.

The algorithm checks to determine whether the user profile is a member of one or more groups. If the profile is not a member of a group, then the algorithm proceeds to the next step, check public authority. If the profile is a member of one or more groups, then each group user profile is processed as described in the check user profile step (Step 3) except for the Quick Check. The quick check is not part of processing group profiles. The authority from the group profiles is additive.

Each group profile is checked and its authority accumulated until the accumulated authority is sufficient or there are no more group profiles. Processing of the group profiles results in the same three cases as for user profiles:

- Authority has been located and is sufficient.
- Authority has been located but is insufficient.
- Authority has not been located.

If authority has not been located, then the algorithm proceeds to check the public authority. If the authority is sufficient, then the algorithm terminates and access is granted. If authority is insufficient, then the algorithm skips to the check program adoption step.

- **Step 5 - Check Public Authority**

  Checking the public authority of an object involves locating the public authority. The public authority can be set to any of the valid authority values. If the public authority is set to a value of *AUTL, then the public authority is obtained from the authorization list.

  If the public authority is sufficient, then the algorithm terminates and access is granted. If the public authority is insufficient, then the algorithm proceeds to the next step to check program adoption.

- **Step 6 - Authority Sufficient Test**

  The authorities that have been located in the user profile, group profile, and public authority are accumulated in a user process authority mask and compared to the required authority. If the authority is sufficient, then checking terminates. Otherwise, the algorithm continues to Step 7, Check Program Adoption.

- **Step 7 - Check Program Adoption**

  Another source of authority for objects is program-adopted authority. Program-adopted authority is authority that is adopted from the owner of the program. Adopted authority is added to any other authority found for the user. Only the authorities of the owner are adopted. If the owner has a group profile, the group’s authorities are not considered. Adopted authority is a program attribute that is specified when the program is created. If program adoption is specified, then the authorities associated with the program owner’s user profile are checked to determine whether authority is sufficient to access the object. The system may use the adopted authority from the original program the user called or from earlier programs in the program stack. If the adopted authority check locates sufficient authority, then access is granted. If the result is insufficient, then access is denied. Program adoption is discussed in section 3.2.3.4 “Process Management” on page 3-61.

6.2.5. Access at Creation

When objects are created, the public authority for that object is stored in the object header (EPA header). This can be explicitly specified at creation or set to the default system value (QCRTAUT).[3] The default authorities for objects created on the system are *ALL authority for the owner and the default system value (QCRTAUT) for *PUBLIC. The user can also set an object’s authority to that of the special value *LIBCTRAUT, which is the public authority for the library containing the object. No primary group is identified for the object and the primary group authority is *NONE.

[3] This is set to *EXCLUDE in the evaluated configuration per Trusted Facilities Manual (TFM).

6.2.6. Granting and Revoking DAC privileges

When objects are created, the user can either specify the public authority to the object or use the default system value. Once the object has been created, the authority to the object can be modified, added, or removed by using either the CL command Edit Object Authority, the CL command Grant Object Authority, or the CL command Revoke Object Authority. Authority is removed by either using the CL command Edit Object Authority or the CL command Revoke Object Authority. Changes in authority take effect on MI instruction boundaries that work with system pointers. In the case of program adoption, changes of authority are effective at invocations. Adopted authority may be checked at each invocation.

Database objects have additional authority restrictions and interfaces. In addition to CL commands, SQL statements exist to grant and revoke authority.

For a logical file, *object authority* to the file can be granted, but because logical files contain no data, *data authority* (see section 6.2.1 “Authorities” on page 6-4) cannot be granted. Providing access to the data defined by a logical file requires a combination of object authority to the logical file and data authority to the associated physical file. If object operational authority is not granted to a user for a physical file, that user cannot open the physical file to access its data.

6.2.7. Propagation of Authority

This section discusses control of propagation of authority via *AUTLMGT and *OBJMGT authorities. To grant authority to an object, the process must be the owner, have *ALLOBJ special authority, or have *OBJMGT (*AUTLMGT for authorization lists). Only the owner or user with *ALLOBJ can grant or modify another user’s *OBJMGT authority (or *AUTLMGT for authorization lists). A process with *OBJMGT can grant or modify any of the authorities they have with the exception of *OBJMGT authority. For example, a process with only *OBJTMGT and *USE to Object A can give other users *OBJOPR, *READ and *EXECUTE but none of the other authorities. The authority available to the process is determined using the algorithm described in section 6.2.4 “Access Control Algorithm” on page 6-7, with one exception. When determining authority that can be granted, only group profiles that have *OBJMGT (*AUTLMGT) have their authority accumulated.

6.3. Special Authority Algorithm

The algorithm for checking a user process’ special authority is similar to the DAC algorithm. Figure 6-6. “Special Authority Algorithm” on page 6-16 illustrates the steps involved in checking for special authorities. The algorithm consists of the following steps:

- Process User Profile
- Process Group User Profiles
- Check Program Adoption

Special authority checking is all additive. The algorithm searches the user profile, the group profiles, and program adoption until sufficient authority is found or the algorithm reaches the last step.

6.3.1. Step 1 - Process User Profile

This step checks the user profile to determine whether the user profile contains special authorities. If the special authorities located in the user profile are sufficient for access, then the algorithm terminates. If special authority is located, but not what is required, then the algorithm goes to the next step, Process Group pro-

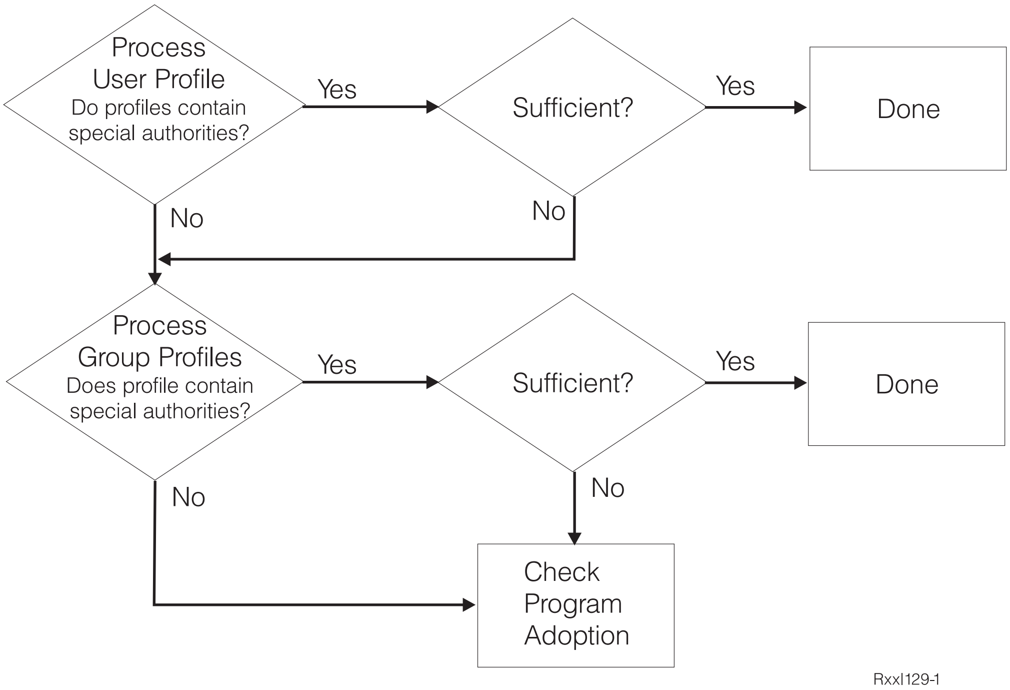


Figure 6-6. Special Authority Algorithm

files. If there are no special authorities in the user profile or if the special authorities are less than what is required, then the group profiles are checked.

6.3.2. Step 2 - Process Group Profiles

Group profiles are processed in the same manner as user profiles. In this step, the profiles that are searched for special authorities are the group profiles, rather than the user profile. Each group profile is searched one at a time until sufficient authority is accumulated or all groups have been tested.

The algorithm checks to determine whether the user profile is a member of one or more groups. If the profile is not a member of a group, then the algorithm proceeds to the next step, check program adoption. If the profile is a member of one or more groups, then each group user profile is processed as described in the check user profile step (Step 1). Each group profile is checked until sufficient special authority is accumulated or all group profiles have been checked. Processing the group profiles results in the same cases as for user profiles:

- Special Authority has been located and is sufficient.
- Special Authority has been located but is insufficient.
- Special Authority has not been located.

If special authority has not been located, then the algorithm proceeds to check the program adoption. If the special authority is sufficient, then the algorithm terminates and access is granted. If special authority is insufficient, then the algorithm checks for program adoption.

6.3.3. Step 3 - Check Program Adoption

Another source of special authority for objects is program adopted authority. Program adopted special authority is special authority that is obtained from the owner of the program. Adopted special authority is added to any other special authority found for the user. Adopted special authority is a program attribute that is specified when the program is created.

If program adoption is specified, then the special authorities associated with the program user profile are checked to determine whether authority is sufficient to perform the function. The system may use the adopted special authority from the original program the user called or from earlier programs in the program stack. Program adoption is discussed further in section 3.2.3.4 “Process Management” on page 3-61.

6.4. Object Reuse

Object reuse concerns the allocation of resources that have been used to store information and then released back to the system for future use. A subject must not be able to scavenge data from resources previously allocated to other subjects. The object reuse policy for AS/400 is two fold:

1. Clear storage before reuse.
2. Restrict access to an object until the resource has been written into.

The paragraphs below discuss object reuse for memory frames, for disk storage sectors (and the objects stored within the disk sectors), for IOP buffers, device buffers, and for removable media.

6.4.1. Memory Frames

A frame of memory is cleared before a new frame is moved in. This ensures that a program does not access data left in memory by another program. Section 4.1.3.1 “Storage Management” on page 4-18 describes how MSM guarantees that disk sectors allocated to a program are over-written from main storage before being read. The section referred to points out that MSM does not guarantee that reallocated frames will be cleared. Therefore, the SLIC component that requested the re-allocated frame must ensure that the frame is cleared before it is read.

6.4.2. Disk Storage Sectors

When a segment is created, the MI instruction specifies parameters to say whether the segment is permanent or temporary; if it is temporary, whether the segment is reusable; and, if reusable, whether to zero the segment before reuse. SM does not reuse segments that have been assigned during the current IPL even if they are freed, unless they are temporary, reusable segments. The default setting for reusable segments is to zero on first reference after being reallocated. Section 4.1.3.1 “Storage Management” on page 4-18 discusses this in more detail.

Management of virtual addresses and single-level storage on the AS/400 system ensures that two objects never share the same address space. It also ensures that an object’s boundaries are not crossed when the system manipulates that object. See section 3.2.3.3 “Storage Management” on page 3-47 for further details.

An object must be created before it is accessed. When an object is deleted, all authority information for that object is deleted with it. If an object with the same name and type is later restored from backup media, the TCB performs checks to determine how public authority and ownership for the restored object should be set. If a user’s authorities to objects are restored on the system, the system also performs checks to ensure that the user’s authorities match the objects actually on the system. See section 4.3.2 “Backup and Recovery” on page 4-90.

The MI instruction Set Access State (SETACST) is used to cause the sectors occupied by deleted MI objects to be cleared before reuse. Sectors from both internal and external objects can be reused. For example, OS/400 reuses some of the job structure for multiple jobs. These objects are cleared at job termination. If the cleanup procedure gets no errors, it places the cleared objects on a reuse queue. If the cleanup procedure encounters an error, the objects are not placed on the reuse queue and therefore are not reused. Examples of these objects include the WCB and the LDA. Section 4.1.3.3.1 “Job Structure” on page 4-24 discusses this further. The design documentation states that such objects are only reused within a job and are never shared between users.

All of database records are stored in physical files. When a physical file is initially created, all of the segments are cleared by the DB. As records are added to a physical file, the segments are populated. As rows are updated and deleted, segments are reorganized as needed. When a record is deleted from a physical file, the actual data is not cleared. The ‘record deleted’ bit in the ‘non-data’ portion of a record is changed to indicate that the record has been deleted and the space cannot be accessed until a new record has overwritten that space. When a new record, to be inserted contains only a subset of the fields in physical file record, the fields not in the new record are given default values. The default values are defined via the ‘DFT’ (Default) attribute when the physical file was created. If the DFT attribute is not used, blanks for character fields and zeros for numeric fields are used as the default values. However, if the allow null (ALWNULL) attribute for the field is specified when the physical file was created, then the character or numeric fields default to the null value.

Data to be printed is placed in a database member of a database file owned by QSPL. After the data is printed, the DB member can be reused for another print job. Clearing of QSPL Database File Members is done by the system job QSPLMAINT. In some other cases, for improved performance, the clearing of the database member can be done within the job deleting the file. This is done by the writer and any batch jobs. section 4.1.3.4 “Spool” on page 4-35 discusses the clearing process.

6.4.3. IOP Buffers, Device Buffers, and Removable Media

The workstation, printer, diskette, DASD, optical, and tape IOPs have buffers, and all of these buffers use an overlay algorithm to ensure that no residual data is sent either to the program or to the device.

Printers and workstation devices have single page buffers. These buffers also use an overlay algorithm to ensure that no residual data is sent either to the program or to the device.

The TFM describes the procedure for degaussing tapes and warns that data may be disclosed if this is not done. The TFM also gives instructions for protecting removable media, especially removable media containing save files.

6.5. Audit

The AS/400 system has the ability to audit all security relevant events. All audit entries are maintained in *journal receivers* (*JRNRCV) in the security audit *journal* (*JRN). The security audit journal serves as a repository of journal receivers that contain the actual audit data. Only authorized users may gain access to the security audit journal and journal receivers; they are system domain objects and are protected by the discretionary access control policy. Furthermore, there are no external interfaces available to modify journal entries or to remove individual entries from the journal. Once the entries have been written to the journal, they cannot be modified. Both preselection and postselection options are used to select and analyze audit data.

A user with the *AUDIT special authority can create the security audit journal and a journal receiver. The security audit journal must exist in library QSYS; and the security audit journal and journal receiver are owned by the user who created them. The journal receiver and the journal have default PUBLIC authority of *EXCLUDE, meaning that only the owner and other trusted users have access to the objects.

This section describes the AS/400 audit mechanism. It starts with some definitions and follows with a description of the audit mechanism and the flow of audit data. Audit entries are discussed next. The section concludes with a discussion of audit data loss.

6.5.1. Background Concepts

The following terms are used throughout this section:

- Audit Session: This is the time from the enable of audit to the corresponding disable.
- Audit Administrator: This is the person with *AUDIT special authority who is responsible for all tasks associated with the maintenance of the audit mechanism, in particular for the audit system values set-up, audit preselection, and audit data reduction.

6.5.2. Implementation of Audit

Auditing of security relevant actions in AS/400 is accomplished by having the software that produces such an action call the appropriate OS/400 or SLIC routines. The auditing routines are functionally the same; only one is used at the OS/400 level and the other below. However, these routines ultimately call the same SLIC routines to append an entry into the journal receiver. These routines determine whether auditing is currently active by testing the QAUDCTL (audit control) system value values *AUDLVL (audit level) and *OBJAUD (object audit) (see section 6.5.4.1 “System Values for Auditing” on page 6-22). The *AUDLVL and *OBJAUD values are set when the system values for auditing are changed. If auditing is active and the security action is preselected, then these routines will send appropriate audit entries to the journal receiver in the security audit journal.

When audit data are received from the auditing routines, they are collected in an audit buffer in main storage. The number of audit data entries that can be collected into the audit buffer before being forced to auxiliary storage is defined by the Audit Administrator and determine the potential amount of audit loss in the event of unexpected system failure. Whenever certain predefined thresholds are met, the buffer is written to auxiliary storage.

When auditing is turned on for many different actions or many objects on the system, SLIC knows when more than one process is waiting to deposit a journal entry into the audit buffer at the same time. SLIC continues to deposit entries into the journal receiver until there are no more processes waiting to deposit an entry or until the audit force level threshold is reached (based on the QAUDFRCLVL system value [see section 6.5.4.1 “System Values for Auditing” on page 6-22]). SLIC then does an *audit force* on the audit buffer for the audit entries deposited in the audit buffer. This audit force causes the audit buffer to be written from main storage to auxiliary storage.

The audit mechanism also monitors the size of the journal receiver in auxiliary storage. If the journal receiver reaches an Audit Administrator-specified threshold, the audit mechanism generates warning messages that are sent to the Audit Administrator. This allows the Audit Administrator to react to possible journal receiver-full conditions and to keep the system from shutting down. The threshold value is specified by the Audit Administrator. The warning messages are sent on an hourly basis.

6.5.3. Contents of Audit Entries

All audit entries are classified by entry type. The entry type defines the format of the audit entry and identifies within the audit entry the specific event that occurred. Each audit entry in the journal receiver contains the following information:

- Length of Entry: Total length of the journal entry
- Sequence Number: Attached to each journal entry
- Journal Code:
  - T for security entries
  - U for user entries (through the CL command Send Journal Entry [SNDJRNE])
  - J for system entries (e.g., IPL of the system)
- Entry Type:
  - AD - Audit changes
  - AF - Authority failure (e.g, discretionary access control, domain check, hardware storage protection)
  - AP - Obtaining authority through a program that adopts
  - CA - Authority change
  - CD - Command string audit
  - CO - Create object
  - CP - User profile changed (e.g., changing a security relevant attribute)
  - CQ - Change of a *CRQD object[4]
  - DO - Delete object
  - DS - DST security password reset
  - GS - Give Descriptor[4]
  - IP - Interprocess communication
  - JD - Change to user parameter of a job description
  - JS - Actions that affect jobs
  - LD - Link, unlink, or look up directory entry
  - ML - Office services mail actions[4]
  - NA - Network attribute changed[4]
  - ND - APPN directory search filter violation[4]
  - NE - APPN end point filter violation[4]
  - OM - Object move or rename
  - OR - Object restore
  - O1 - Optical Access - Single File or Directory
  - O2 - Optical Access - Dual File or Directory
  - O3 - Optical Access Volume
  - OW - Object ownership changed
  - PA - Program changed to adopt authority
  - PG - Change of an objects primary group
  - PO - Printed output

[4] These entry types are not generated in the C2 configuration. They are included for completeness.

- PS - Profile swap
- PW - Invalid password
- RA - Authority change during restore
- RJ - Restoring job description with user profile specified
- RO - Change of object owner during restore
- RP - Restoring adopted authority program
- RQ - Restoring a *CRQD object[4]
- RU - Restoring user profile authority
- RZ - Changing a primary group during restore
- SD - Changes to system distribution directory[4]
- SE - Subsystem routing entry changed
- SF - Actions to spooled files
- SM - Systems management changes
- SO - Security Object Changes[4]
- ST - Use of service tools
- SV - System value changed
- VA - Changing an access control list[4]
- VC - Starting or ending a connection[4]
- VF - Closing server files[4]
- VL - Account limit exceeded[4]
- VN - Logging on and off the network[4]
- VO - Validation List Actions
- VP - Network password error[4]
- VR - Network resource access[4]
- VS - Starting or ending a server session[4]
- VU - Changing a network profile[4]
- VV - Changing service status[4]
- YC - Document library object accessed (change)[4]
- YR - Document library object accessed (read)[4]
- ZC - Object accessed (change)
- ZM - SOM Method Access[4]
- ZR - Object accessed (read)

- Date of Entry: The system date the entry was made
- Time of Entry: The system time the entry was made
- Name of Job: The name of the job that caused the entry to be generated
- User Name: The user profile associated with the job
- Job Number: The job number
- Program Name: The name of the program that made the journal entry
- System Name: The name of the system

In addition to the above standard fields, there are additional specific fields in log entries based on the entry type. For example, the PW entry type has a *Device Name* field that displays the name of the device on which the password or user profile name was entered. Detailed tables showing the different audit entries are found in the TFM.

6.5.4. Audit Action Preselection

The Audit Administrator can set the audit preselection criteria for an Audit Session. There are three auditing levels that the administrator can choose from:

- System-wide auditing that includes all users and objects
- User profile auditing for individual users
- Object auditing for specific objects

6.5.4.1. System Values for Auditing

System values are used to turn auditing on and select system-wide security relevant actions. The system values to control security auditing on AS/400 are as follows:

- QAUDLVL: Determines which security related actions are logged to the security audit journal. This level of auditing is independent of any object auditing that is active on the system. One or more of the following values may be specified. If *NONE is specified, it must be the only value specified.
  - *NONE: No events are logged on a system-wide basis. Actions are logged for individual users based on the AUDLVL value in their user profiles.
  - *AUTFAIL: Failed sign-on attempts and authority violation on accessing an object are logged.
  - *PGMFAIL: System violations are logged (e.g., domain violation, hardware storage protection violation).
  - *SECURITY: Change in security system values, user profiles, or authority to objects are logged.
  - *DELETE: Deletion of objects are logged.
  - *SAVRST: Saving objects from the system and restoring objects to the system are logged.
  - *CREATE: Creation of objects are logged.
  - *OBJMGT: Renaming and moving of objects are logged.
  - *OFCSRV: OfficeVision for OS/400 tasks are logged.
  - *OPTICAL: Operations on optical devices are logged.
  - *NETCMN: APPN filter support violations are logged.
  - *JOBDTA: Actions that affect jobs are logged (e.g., stopping a job).
  - *PGMADP: Running programs that adopt authority are logged.
  - *PRTDTA: Printing spooled files are logged.
  - *SERVICE: Use of system service tools are logged.

  - *SPLFDTA: Actions on spooled files are logged (e.g., sending).
  - *SYSMGT: Use of system management functions are logged (e.g., changing power on/off schedule).
- QAUDCTL: Turn auditing on or off at the audit level or at the object level. It functions as a master switch to turn auditing on or off. The possible system values for QAUDCTL are:
  - *NONE: No auditing is performed.
  - *AUDLVL: Audit function selected by QAUDLVL (this value must be specified in order for any audit entries to be generated for functions identified by the QAUDLVL system value) and the AUDLVL attributes for user profiles.
  - *NOQTEMP: No auditing of objects in the job temporary library QTEMP is performed.
  - *OBJAUD: Audit of object access is performed.
- QCRTOBJAUD: Default audit specification for new objects. The possible system values for QCRTOBJAUD are as follows:
  - *NONE: No auditing is done for new objects.
  - *USRPRF: Audit new objects based on user profile settings (e.g., audit an object when user1 is accessing it but not when user2 is accessing it).
  - *CHANGE: Audit change operations for new objects.
  - *ALL: Audit all operations for new objects.
- QAUDENDACN: Specify actions when the system is unable to write audit entries. The possible values are as follows:
  - *NOTIFY: Inform administrator (hourly notification).
  - *PWRDWNSYS: Terminate processing and power the system down.
- QAUDFRCLVL: Controls forcing audit data to auxiliary storage. The possible values are as follows:
  - *SYS: Allow system to select forcing of audit data based on internal system processing.
  - Number: Number 1-100 of entries prior to forcing audit data.

6.5.4.2. Auditing of Actions

The QAUDCTL (audit control) system value, the QAUDLVL (audit level) system value, and the AUDLVL (action auditing) parameter in user profiles work together to control action auditing:

- The QAUDCTL system value starts and stops action auditing.
- The QAUDLVL system value specifies which actions are audited for all users of the system.
- The AUDLVL parameter in the user profile determines which actions are audited for a specific user. The values for the AUDLVL parameter apply in addition to the values for the QAUDLVL system value.

6.5.4.3. Auditing of Objects

The QAUDCTL system value, the OBJAUD value for an object, and the OBJAUD value for a user profile work together to control object auditing. The OBJAUD value for the object and the OBJAUD value for the user who is accessing the object determine whether a specific access should be audited. Table 6-1 on page 6-24 shows how the OBJAUD values for the object and user profile work together.

Table 6-1. How Object and User Auditing Work Together

	OBJAUD value for user		
OBJAUD value for object	***NONE**	***CHANGE**	***ALL**
*NONE	None	None	None
*USRPRF	None	Change	Change and Use
*CHANGE	Change	Change	Change
*ALL	Change and Use	Change and Use	Change and Use

6.5.5. Auditing of Database Record Modifications

In addition to the audit preselection criteria, that an Audit Administrator can select, as listed in section 6.5.4 “Audit Action Preselection” on page 6-22, the administrator can audit the actual record changes by journaling the physical file that contains the records. The administrator can create a journal receiver and attach it to a physical file. Once a journal receiver it attached, record changes are logged in the receiver. The audit entry contains information about the identity of the user that caused the change, the record identifier, and the pre and post-record modification values. The exact format of the audit entries is described in the [3] Advanced Backup and Recovery Guide, which is part of the Trusted Facility Manual.

6.5.6. Analyzing Audit Data

The Audit Administrator can analyze all audit data entries using one of two methods:

- Viewing selected entries at the display device via CL commands
- Writing audit data to a database file and then using a query tool (e.g., SQL) to analyze the entries

The AS/400 system is shipped with “model” database files for each audit journal entry type. The model data base files contain definitional information regarding the structure and format of each journal entry type.

When the Audit Administrator is ready to analyze audit data, he or she may issue the CL command Create Duplicate Object (CRTDUPOBJ) to duplicate the model data base file for the journal entry type to be analyzed. Once the duplicate object is created, then the Audit Administrator may issue the CL command Display Journal (DSPJRN) to extract the journal entries from the journal and write the entries to the duplicate database file. This command opens the duplicate data base file (via data management) that was specified on the OUTFILE parameter of DSPJRN. Then each audit entry in the journal was written to the database file as a separate record in that file.

The TFM provides guidelines and instructions to the administrator on how to analyze audit data.

6.5.7. Audit Data Loss

Audit data loss can occur under three types of circumstances:

1. When the journal receiver runs out of space[5]
2. When a disk drive crashes
3. When the system crashes

The audit mechanism monitors the availability of free space in the journal receiver and notifies the administrator when the journal receiver is full (no further auditing takes place), unless the administrator has configured the system to shut down instead.

Furthermore, the security audit journal can have two journal receivers. Dual receivers cause the system to write the same journal entry to each receiver when they are attached to the journal. This support is given to allow journal receivers to be created into different auxiliary storage pools to prevent data loss if a disk drive fails. Therefore, the administrator can prevent the loss of audit data by disk drive failure conditions.

A system crash causes the loss of audit data from the journal receiver in main storage. The number of audit entries that can be collected in the journal receiver in main storage can be configured by the Audit Administrator.

When the system is up and running and the audit journal receivers are full, these journal receivers changed without loss of audit records.

While the changing of journal receivers is in progress, auditing on the system can still be running. Any job that is attempting to send an audit record to the auditing journal waits until the change of the journal receiver is finished. This wait is accomplished by waiting for a lock on the journal object. If the wait on the lock times out, then the sending of the journal entry is retried until it is successful.

[5] The maximum size of the journal receiver is 2 GB.

Section 7
Assurance

7.1. System Architecture

The AS/400 TCB uses hardware and software mechanisms to provide a user state and a system state as is described in the paragraphs immediately below. In addition, the other mechanisms that support TCB isolation and protection are also described. The resources controlled by the TCB are listed in Table 5-3 on page 5-10. The mechanisms providing resource isolation and protection are described in the final paragraphs below.

7.1.1. TCB Isolation and Protection

TCB isolation and protection are provided by state and domain attributes, by blocked MI instructions, by control of the translator, by save and restore restrictions, by hardware storage protection, and by tagged pointers.

7.1.1.1. State and Domain Attributes

The section 4.1.2.2 “Programmatic Interfaces” on page 4-15 discusses the various state and domain combinations. In addition, section 2.3.1 “TCB Structure and Definition” on page 2-7 describes how the state and domain attributes are used to separate TCB executable code from non-TCB code.

The program state indicates how the process state should be set when the program is run. Both user state and system state programs can run in the same process. When a program is called, the SLIC reads the program state attribute and changes the state of the process, if required, and sets the machine state register (MSR bit 56) appropriately. On return, the original process state is restored. Users, even trusted users, have no way to affect this state change.[1]

Users cannot cause the program state attribute to be set to anything other than User State. There are no interfaces that allow the program state attribute to be modified. Only IBM developers, using a special tool not shipped as part of the evaluated configuration, can create a program object with the state attribute set to system state or inherit state. The translator shipped as part of the evaluated configuration sets the state attribute only to user state.

To limit user program access to objects, all objects are assigned an attribute called the object domain. The domain of an object is determined when the object is created. No objects are created without the domain being specified. The object domain checks are enforced for all operations which use a system pointer to access the object.

[1] The service tools can change this attribute, assuming that the service tool user could locate the appropriate bits. However, the service tools are not used in the evaluated configuration.

The domain of an object can be either user or system. Objects in the user domain can be accessed by either a user-state or a system-state program with the appropriate permissions. Objects in the system domain can be accessed only by system-state programs with the appropriate authority.

7.1.1.2. Blocked MI Instructions

Some instructions are blocked, and these are analogous to machine instructions that can only be executed while the machine is in supervisor mode. MI instructions can be blocked at translate time or at runtime. Instructions that are blocked at translate time are those that the translator will not translate. Most blocked instructions are also blocked at runtime in user state. Section 3.2.2.2.2 “Partially Blocked” on page 3-40 discusses those MI instructions that are unblocked at translate time, but are partially blocked in user state at runtime.

Section 3.2.3.10.6 “The Router” on page 3-105 describes the use of the router table, which includes the blocked MI instructions. Since this table cannot be altered (it is not addressable by any MI program, and there are no interfaces available to alter it), new MI instructions cannot be added to it by any tool in the evaluated configuration.

7.1.1.3. Control of Translator

The translator is called by the MI instructions Create Program (CRTPG) and Create Module (CRTMOD). Only the translator described in section 3.2.3.10 “MI Translator” on page 3-101 is included in the evaluated configuration. Even if a user were to write a substitute translator, or buy one of the commercially available non-IBM translators, the Trusted Facility Manual (TFM) warns that introducing such a translator removes the system from the evaluated configuration. Further, such a translator cannot create an encapsulated program object because MI programs can write data only into spaces and spaces cannot have the program MI object type.

A program object has a verifiable internal format. There is no edit operation defined or allowed for an object of this type, and tools capable of making patches to RISC code are not allowed in the evaluated configuration.

7.1.1.4. Save and Restore Restrictions

See section 4.3.2.4 “Validation of Programs Being Restored” on page 4-94 for how the restore function supports the system architecture. The program validation technique described in that section helps the system administrator to detect the use of the restore function to introduce programs into the system that are not allowed on the evaluated configuration. These programs might have been compiled on a non-evaluated configuration using a third-party translator, or may have been altered by use of the Dedicated Service Tools (DST).

7.1.1.5. Hardware Storage Protection

Hardware storage protection is introduced in section 3.1.1.1.3 “Hardware Storage Protection” on page 3-10 and further discussed in section 4.1.3.1 “Storage Management” on page 4-18. When an object is created, the hardware storage protection attribute for the object is specified. The hardware storage protection attribute can be one of the following values:

- No read or write from user state
- Read-only from user state
- Read-only from either user or system state
- Read and write from either user or system state

When an object is paged in (loaded from auxiliary storage into main storage), the object storage protection attribute is set for each page making up the object. During execution of each RISC instruction, the hardware determines whether the page frame is hardware storage protected. In this way, user state programs can have hardware storage protection read-only access to objects such as the entry point table.

Hardware storage protection is also important when a subject obtains a space pointer to the associated space of an object. The hardware storage protection values prevent a user state program from modifying the associated space of an object to which the subject has only read access.

7.1.1.6. Tagged Pointers

The section 3.1.1.1.2 “Tag Bits” on page 3-8 introduces the hardware tag bits set to identify a valid pointer data object. Section 3.2.1.2.4 “MI Pointers” on page 3-21 discusses the various types of pointers, as well as the MI instruction Resolve System Pointer. Obtaining a pointer gives a process addressability to an MI object. Any attempt to modify a tagged pointer, except with the proper MI instruction, causes the tag bit to be cleared and the pointer to become invalid. This, along with translator control of addressability to space pointer machine objects, prevents non-pointer data from being used as pointer data.

7.1.2. Resource Isolation and Protection

Resource isolation is provided by object encapsulation, by the use of the domain attribute along with DAC and audit requirements, by hardware storage protection as noted above, by validating parameters passed between user state and system state programs, by preventing modification of internal control blocks, and by restricting message handling between user and system state programs.

7.1.2.1. Object Encapsulation

The section 3.2.1 “Object Management in SLIC” on page 3-17 describes object encapsulation for MI objects. MI objects are provided at the machine interface and have a defined format consisting of an encapsulated, functional part and an associated space. The encapsulated, functional part can be modified only through the MI instructions provided by the object manager for that object. Object encapsulation prevents modification of the program state attribute, the object domain attribute, and the hardware storage protection attributes.

7.1.2.2. Domain Attribute

The section 5.3.3 “DAC Policy” on page 5-9 presents the general policy for named objects. Named objects are created in system domain, and DAC and audit requirements are enforced on these objects. Named objects created in system domain are accessible by user state programs only through the MI instruction set. The exceptions to this general rule are as follows:

- Objects of type *PGM and *SRVPGM can be in either domain but, on the evaluated configuration, can be created only in user domain. The user-created *PGM object has a hardware write-protected associated space.
- Objects of type *USRIDX, *USRQ, and *USRSPC can be created in system domain. If they are created in user domain, the object is forced into the job's QTEMP library in the evaluated configuration. Addressability to the QTEMP library is confined to the user's job. No interface exists by which a user can obtain a system pointer to another user's QTEMP or to the objects within it. Any object placed in QTEMP is not shared. See section 5.3.1 "Object Determination Approach" on page 5-6 for more on non-shared objects.

7.1.2.3. Validating Parameters

When parameters are passed between user state and system state programs, both the input and output parameters must be validated to ensure that an invalid parameter value cannot cause the TCB to perform some function which the user state program is not allowed to perform, or that an invalid parameter does not contain a value that would unintentionally modify a system domain object. Parameter validation prevents disruptive function failures. Even parameters that are not used by the UDSS program but are passed through to a SDSS program are verified by the UDSS program, according to the design documentation. The parameter validation mechanism applies to all callable APIs, all command processing programs, compiler execution time subroutines, and data management services. Parameter validation is described in section 4.1.2.2.4 "User Domain System State Programs" on page 4-16.

When a UDUS program receives a message from an external source (*EXT), any pointers in the message replacement text are removed.

7.1.2.4. Preventing Modification of Internal Control Blocks

In the evaluated configuration, internal control blocks, or internal objects in AS/400 terminology, may be in user domain or in system domain and are protected by the hardware storage protection mechanism. These internal control blocks include the process static, heap, and dynamic storage working areas. Most internal control blocks are in system domain (or are protected by the equivalent hardware storage protection value) and accessible only through blocked MI instructions. An example of a system domain internal object accessible only through blocked MI instructions is the AUT. A system value is an example of a system domain internal object that is visible to users, although still accessible only through blocked MI instructions. An example of a user domain internal object is the entry point table (EPT). User domain internal objects are read-only hardware storage protected. (See section 4.1.1.2 "Internal Objects" on page 4-12 for more about this.)

7.1.2.5. Restricting Message Handling

The following applies to message handling for the evaluated configuration:

- Any user state program can send a message of any type to any other user state program.
- Any system state program can send a message of any type to any user or system state program.
- A user state program can send a non-exception message to any system state program.
- A user state program can send an exception type message (status, notify, or escape) to a system state program if either the system state program is a request processor, or the system state program called a user state program. The user state program sending the exception message does not have to be the user state program called by the system state program.

7.2. System Integrity

The following system hardware integrity tests are included with the evaluated configuration.

7.2.1. Power-On/IPL Hardware Diagnostics

As the machine is powered on, the IOP that runs the service processor (SP) code and other IOPs run a series of common tests that are stored in the Flash memory on each IOP card. These tests include exercising the instruction set of the processor on the IOP, validating that the processor can detect exception conditions (e.g., divide by zero), testing that cache, and then using the cache to test the memory on the IOP card. Once these tests have successfully completed, the IOP signals that it is ready.

Once the SP is finished running its internal tests, it walks the I/O bus on which its IOP is located to determine what other IOPs are configured on that I/O bus. As it walks the bus, the SP tests the IOPs externally by sending messages to the IOPs and receiving responses from the IOPs. The SP then locates the load source. The load source is usually attached to the IOP on which the SP code runs, but it may be located on another IOP on this I/O bus. Once the load source is located, additional SP code is loaded into the IOP memory from the load source.

This additional SP code runs tests on the rest of the IOP and I/O Adapter (IOA) hardware on this bus. Then the AS/400 processor hardware chips are tested with a built-in self test technique in which a pattern of bits is serially loaded into a chip and the chip is clocked a few hundred thousand times. A signature pattern is then compared to the expected results. Chip arrays are also tested by array control logic that verifies all addresses and data bits with a pass/fail status that the SP checks. The I/O driver/receiver intercommunications of the chips and cards are tested via an A/C wire test. Tests are run to determine the functionality of the AS/400 main memory and caches. The DMA facilities of the IOPs and system buses are also verified during IPL.

Once these tests are complete, the SP loads the SLIC nucleus from the load source into the AS/400 main memory. The SP initializes the processor, switches control of the I/O bus over to SLIC, and starts the AS/400 processors running. Configuration, initialization and testing of additional I/O buses, IOPs, and IOAs is handled by SLIC.

7.3. Ratings Maintenance Phase

IBM will be participating in the Ratings Maintenance Phase (RAMP). IBM has a configuration management plan for AS/400, which is described in the document titled *AS/400 Rating Maintence Phase - Process* [16].

The RM-Plan is managed by the Responsible Corporate Officer (RCO), who is also the Software Development Manager for the AS/400. The RCO designates VSA assignments and responsibilities, communicates with the NSA on administrative issues when necessary, and provides corporate assurance that the RM-Plan and submissions of evidence accurately conform to the RAMP Process Document.

A lead VSA appointed by the RCO is responsible for the coordination of other appointed VSAs and their activities. The lead VSA is also responsible for the execution of all technical tasks in RAMP, including presentation and defense of AS/400 RAMP evidence.

IBM uses a configuration management system called Integrated Development Support System (IDSS), which is a tool for documenting, communicating, versioning, and scheduling changes. IDSS manages changes to configuration items listed in Appendix A, “Evaluated Hardware”, on page A-1 and Appendix B, “Evaluated Software”, on page B-1. To supplement IDSS, IBM uses the Configuration Management Version Control (CMVC) system for development, maintenance, and code versioning for C, C++, and new internal programming languages. IDSS is still the ultimate control mechanism for integration of code into AS/400; however, IBM plans to gradually shift central control from IDSS to CMVC.

The configuration items are separated into two categories due to the different granularity levels. The first category is software, which includes OS/400 (and products) and SLIC. All software is divided into components, a component is a logical grouping of functionality that consists of parts. A part consists of source code, executables; headers, test case source, design documentation, and user documentation. The second category is hardware, which includes all devices in the TCB. Each hardware device has a device type associated with it and each device type has different models. All models associated with a device type are architecturally equivalent.

IDSS implements a process known as Design Change Request (DCR), which is the mechanism for defining and managing changes to the AS/400 software. The contents of the DCR is a reflection of the work that must be completed by all groups in the development of the software from the time design work is begun until the software is ready to ship to customers. During the development of software, DCRs assume different status. The procedures that are used to move the DCRs through each status is a means of ensuring that all appropriate work is completed and, if required, the correct information is provided. A DCR contains the following sections; approvals, related DCR numbers, brief description, external interfaces, internal interfaces, performance specifications and design points, security/integrity, along with implementation notes and compatibility. Each DCR must be presented at the DCR Technical Review meeting, which ensures product/system consistency, compatibility, technical feasibility, correctness and security/integrity. The System Design Control Group (DCG) is the controlling body of which a VSA is a member. A VSA performs the security analysis on each DCR. During the security analysis process, the VSA assesses the DCR for relevance and impact with regard to architecture, discretionary access control, audit, identification and authentication, object reuse, integrity, modification, delay, and denial of service. This is done by analyzing the contents of the DCR (including the security questionnaire required for each DCR), talking to the developers related to the DCR, and reviewing the results of the security analysis with the DCG to ensure correctness. The security questionnaire is designed such that any change to a TCB interface or a change in the philosophy of protection will be noted such that further analysis can be done. The questionnaire consists of 17 basic questions dealing with security aspects of both XPF and SLIC; each question then may have sub-questions. The security aspects of the questions deal with new objects and changes to existing objects relative to discretionary access control, state, domain, object reuse, audit, identification and authentication. Once security analysis is complete, a DCR is categorized based upon the analysis as either security relevant or non-security relevant. The security relevant changes are further categorized as to major or minor security relevance. Minor security relevant changes are components that have minor security relevance (i.e., parsers, data collectors, interpreters, etc.)

and major security relevant changes are components that have security relevance (i.e., any component that manages an object, communicates between processes, determines authorization, etc.).

If a new test case is required, then a documentation change to add the new test case to the current (new one created for each release) C2 Test DCR is made. The documentation describing the test case variation objectives and test case expected results is also completed and indicated in the C2 Test DCR. If any new documentation that describes the change being made is needed, it is added to the appropriate section in the RM-Plan. Changes to the AS/400 TCB are taken through the RAMP procedures only at a release boundary. These changes can either be in the form of new function, enhanced function, bug fixes (from previous releases) or documentation changes. A bug in the AS/400 software is reported via an Authorized Program Analysis Report (APAR) and a fix is made available (if accepted) via a Program Temporary Fix (PTF). Since the AS/400 submits RAMP evidence only at a release boundary, a site is not allowed to install a PTF until that PTF has been through the RAMP cycle.

A release is a separate entity but is based upon a previous release, plus any DCRs. This provides the ability for a specific release level to be maintained and regenerated upon request. A PTR for the next release is created for every APAR accepted on a current or previous release. This allows bug fixes to be integrated into the next release.

As part of the component/regression test for a release, all test cases that are listed in the C2 Test Plan are run. If the test case is in source format, it is recompiled and placed on a C2 test system and executed. If the test case has executable code, the code is restored onto the C2 test system and executed. In either case these are manual steps taken. When the test case achieves the expected results, the C2 Test DCR is updated to reflect that the test completed successfully. Release exit criteria requires that, after the final build process is complete, the C2 test cases are rerun and each test case obtain the expected results.

When a release is complete, an internal audit is done as outlined by chapter 12 of the *AS/400 Rating Maintence Phase - Process* [16] document. A VSA performs the audit and verifies that all DCRs are in “COMPLETE” status, all test cases have achieved expected results, and all the documentation changes are complete. The audit also consists of a random sampling of DCRs to verify that the change was accepted and placed into plan, security analysis is complete, the VSA has approved, documentation analysis is complete, and test case analysis is complete.

Once the audit is complete, the Rating Maintenance Report (RMR) is generated to summarize the evidence that C2 level of trust has been maintained. The RMR contains a summary of all changes since the previous RAMP level product, description of all security relevant changes and the security analysis of the changes (sections taken from the DCR), and a description of how the RAMP release meets the interpretations that have been issued by the NSA.

IBM currently has a lead VSA and several supporting VSAs assigned to AS/400 in SLIC, OS/400, and testing areas and is committed to the RAMP process.

Section 8
Evaluation of Application System/400 Against the Applicable Requirements

8.1. Security Policy

8.1.1. Discretionary Access Control

Statement of Requirement

The TCB shall define and control access between named users and named objects (e.g., files and programs) in the ADP system. The enforcement mechanism (e.g., self/group/public controls, access control lists) shall allow users to specify and control sharing of those objects by named individuals or defined groups of individuals, or by both, and shall provide controls to limit propagation of access rights. The discretionary access control mechanism shall, either by explicit user action or by default, provide that objects are protected from unauthorized access. These access controls shall be capable of including or excluding access to the granularity of a single user. Access permission to an object by users not already possessing access permission shall only be assigned by authorized users.

Assessment of Product Compliance

AS/400 enforces a Discretionary Access Control (DAC) policy, as described in section 6.2 “Discretionary Access Control” on page 6-4. The DAC mechanism allows named users to define and control access to named objects through the use of authorities. Named users are those users with user or group profiles, as described in section 5.2 “Subjects” on page 5-2. The DAC policy is to perform DAC checks as part of each MI instruction that accesses an object with a system pointer (see section 3.2.2 “Machine Interface” on page 3-39). Additional checks are performed as part of processing CL commands and APIs.

AS/400 defines and controls access between subjects and named objects (see section 4.1.1.1 “External Objects” on page 4-2). The DAC enforcement mechanisms, authorities, authorization lists, and special authorities allow a user to specify and control sharing of protected objects between named individuals, groups, or both. External objects are protected by default; when they are created, only the owner has authority to the object. Trusted users with appropriate special authority also have access to the object. AS/400 provides public authority (*PUBLIC). The *PUBLIC authority default in the C2 system is *EXCLUDE.

The group profile mechanism, described in section 5.2 “Subjects” on page 5-2, allows users to belong to groups.

Controls on the propagation of authority for named objects are discussed in section 6.2.7 “Propagation of Authority” on page 6-15.

Authority and authorization list mechanisms provide the capability to include or exclude access to the granularity of a single user. Access permission to an object by users not already possessing access permission can be granted only by a user with the appropriate authority (object owner, user with *OBJMGT or *AUTLMGT authority).

The Authorization List mechanism provides the capability to group a set of named objects on a list. User and Group profiles may point to the authorization list with their respective authority to all objects on the list.

For this RAMP action, the VSAs evaluated all new and changed function and ensured that the DAC policy was properly designed.

Conclusion

AS/400 satisfies the C2 Discretionary Access Control requirement.

8.1.2. Object Reuse

Statement of Requirement

All authorizations to the information contained within a storage object shall be revoked prior to initial assignment, allocation or reallocation to a subject from the TCB's pool of unused storage objects. No information, including encrypted representations of information, produced by a prior subject's actions is to be available to any subject that obtains access to an object that has been released back to the system.

Assessment of Product Compliance

Storage objects for the AS/400 TCB are memory frames, disk storage sectors, cache, IOP buffers, printer buffers, workstation buffers, optical, and tape drive buffers. Memory frames, disk storage sectors, and cache are cleared before allocation. For IOP and device buffers, the length of the data placed in the buffer is recorded and attempts to read beyond that length are not allowed. For more information about object reuse, see section 6.4 “Object Reuse” on page 6-17.

For this RAMP action, the VSAs evaluated all new and changed function and ensured that all object reuse conformed to policies described in section 6.4 “Object Reuse” on page 6-17.

Conclusion

AS/400 satisfies the C2 Object Reuse requirement.

8.2. Accountability

8.2.1. Identification and Authentication

Statement of Requirement

The TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate. Furthermore, the TCB shall use a protected mechanism (e.g., passwords) to authenticate the user's identity. The TCB shall protect authentication data so that it cannot be accessed by any unauthorized user. The TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user. The TCB shall also provide the capability of associating this identity with all auditable actions taken by that individual.

Assessment of Product Compliance

The security component of OS/400 is responsible for the identification and authentication of users before they are allowed to use the system. This identification is accomplished via user names. Each user has a unique user name. This applies to system operators, security officers, and end users.

The IBM AS/400 uses passwords to authenticate users. At logon, each user is required to supply a user name and password. Password data are stored in an encrypted form in the Authorized User Table (AUT). The AUT is an internal system domain object; therefore, it cannot be accessed by untrusted users.

The TFM recommends that passwords, which are user chosen, have at least five characters. By using system values, the security administrator can control the password lifetime and password composition rules to eliminate trivial passwords.

Since each subject has a unique user name, individual accountability is enforced. Each user name is used to associate all auditable actions with the subject represented by that user name.

For more information about identification and authentication, see section 6.1 “Identification and Authentication” on page 6-1.

For this RAMP action, the VSAs ensured that the validation list (*VLDL) external object, which is an enabler for application-provided user identification functions, does not interfere with or circumvent the system identification and authentication functions.

Conclusion

AS/400 satisfies the C2 Identification and Authentication requirement.

8.2.2. Audit

Statement of Requirement

The TCB shall be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects. The audit data shall be protected by the TCB so that read access to it is limited to those who are authorized for audit data. The TCB shall be able to record the following types of events: use of identification and authentication mechanisms, introduction or objects into a user’s address space (e.g., file open, program initiation), deletion of objects, and actions taken by computer operators and system administrators and/or system security officers, and other security relevant events. For each recorded event, the audit record shall identify: date and time of the event, user, type of event, and success or failure of the event. For identification/authentication events the origin of request (e.g., terminal ID) shall be included in the audit record. For events that introduce an object into a user’s address space and for object deletion events the audit record shall include the name of the object. The ADP system administrator shall be able to selectively audit the actions of any one or more users based on individual identity.

Assessment of Product Compliance

The IBM AS/400 TCB has the capability to audit all security relevant events, including access to objects. Audit data are stored in journal receivers in the security audit journal. The journal receivers and security journal are system domain objects and are protected from unauthorized access and destruction by the discretionary access control mechanisms.

The AS/400 audit mechanism identifies 38 different types of audit events. These events include login and logout, object access, deletion of objects, and actions taken by the administrators and operators, along with other security relevant events, such as modifying system values and changing security-relevant user profile attributes. A complete list of the event types may be found in section 6.5 “Audit” on page 6-18. For each event, the audit record includes the date and time of the event, the user name of the user that initiated the event, and the type of the event. Login/logout events also include identification of the display station used.

The data recorded for events that pertain to manipulation of TCB objects include identification of the object manipulated.

AS/400 provides the Audit Administrator with mechanisms (e.g., SQL, Query/400, etc.) both to preselect and postselect events. Events may be selected based on the event type, object manipulated, and user involved in the action. For more information about audit, see section 6.5 “Audit” on page 6-18.

For this RAMP action, the VSAs ensured that new audit records were logged for new events added to the system.

Conclusion

AS/400 satisfies the C2 Audit requirement.

8.3. Assurance

8.3.1. Operational Assurance

8.3.1.1. System Architecture

Statement of Requirement

The TCB shall maintain a domain for its own execution protects it from external interference or tampering (e.g., by modification of its code or data structures). Resources controlled by the TCB may be a defined subset of the subjects and objects in the ADP system. The TCB shall maintain process isolation through the provision of distinct address spaces under its control.

Assessment of Product Compliance

The TCB maintains a domain for its own execution, and separates this domain from the user domain, by a combination of the state and domain attributes implemented in software. The TCB runs in system state; all other code runs in user state. User state processes cannot access system domain objects directly; instead, they must use the defined TCB interfaces.

Manipulation of the state and domain attributes requires use of blocked MI instructions. Code written on the evaluated configuration cannot use the blocked MI instructions because the translator in the evaluated configuration does not translate blocked MI instructions. Code written on the evaluated configuration cannot issue RISC instructions directly; no way to compile such code exists for the evaluated configuration. The TFM provides procedures for the system administrator to use to guard against object code being restored to the system without retranslation. The hardware cannot be directly accessed.

The MI instruction set provides an object-based interface. This object-based interface enforces discretionary access control and audit for objects. The objects on the evaluated configuration are accessed through system pointers; these system pointers are protected against improper modification by hardware tags. Hardware storage protection bits also protect objects in main storage from improper access or modification by user state processes.

For more about system architecture, see section 7.1 “System Architecture” on page 7-1.

For this RAMP action, the VSAs ensured that new MI instructions were either blocked or unblocked, as appropriate. They also ensured that new programs added to the TCB execute with the correct state and have the proper domain assigned to them.

Conclusion

AS/400 satisfies the C2 System Architecture requirement.

8.3.1.2. System Integrity

Statement of Requirement

Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB.

Assessment of Product Compliance

As part of the IPL process, AS/400 performs various basic assurance tests on the system hardware and firmware. These tests ensure that the system hardware and firmware are functioning properly. If problems are detected, the faulty hardware is labeled as such in a configuration file and error log, and the IPL continues. In some cases, the faulty hardware prevents the system from completing the IPL process. Section 7.2.1 “Power-On/IPL Hardware Diagnostics” on page 7-5 describes the testing that occurs during an IPL.

In addition to the extensive testing that takes place at IPL, IBM provides a set of diagnostic utilities, which are capable of detecting problems in OS/400, SLIC, and some hardware devices. This set of utilities is called system service tools (SST) and dedicated service tools (DST). Both utilities are menu driven, and are outside the evaluated configuration due to the lack of auditing capability. Once the system has been taken out of the evaluated configuration, an administrator can use these tools to trace SLIC components, and MI processes, analyze and perform recovery of disk devices, display the hardware configuration, and trace I/O processors (both workstation and storage). These utilities are restricted from untrusted users by the DAC mechanism as well as requiring special authorities. Trusted users are instructed not to use these utilities while the system is running in the evaluated configuration.

For this RAMP action, the VSAs ensured that adequate diagnostics existed to detect whether the system is operating as designed.

Conclusion

AS/400 satisfies the C2 System Integrity requirement.

8.3.2. Life-Cycle Assurance

8.3.2.1. Security Testing

Statement of Requirement

The security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation. Testing shall be done to assure that there are no obvious ways for an unauthorized user to bypass or otherwise defeat the security protection mechanisms of the TCB. Testing shall also include a search for obvious flaws that would allow violation of resource isolation, or that would permit unauthorized access to the audit or authentication data. (See the Security Testing guidelines.)

Assessment of Product Compliance

IBM's C2 testing effort focus is placed on interfaces into the TCB. Each interface is examined to determine if or how the following policies apply:

- DAC – ensure users may only access objects for which they have authority, as specified by the authorization list, or by use of special authority and adoption.
- Audit – verifies that security relevant events are auditable.

IBM's C2 testing also focuses on testing the system in general for the following:

- Object Reuse – verifies that reusable objects are cleared before they can be used by another user.
- I&A – verifies that only authenticated users are given authorized access to the system.
- System Architecture – verifies that system objects are isolated from untrusted users and that hardware diagnostics are performed during IPL.

For the DAC tests, IBM's philosophy is to use the system groupings for DAC authorities, not the individual authorities (see section 6.2.1 "Authorities" on page 6-4). These system groupings are the access modes that are defined in an authorization list and are the modes users typically assign other users. The approach to testing DAC is as follows: Users are given the minimum authority required to access an object and then demonstrate they have access to that object. The test then attempts access with the user's authority just below the minimum required for that object, which demonstrates that the DAC mechanism is implemented correctly for that given interface. For the minimal authority and just below minimal authority tests, IBM uses the IBM-supplied authority groupings or the individual authorities (see the *V4R1 AS/400 Test Plan for the Department of Defense C2 Evaluation* [21]). For interfaces that work on multiple objects of the same type, IBM does not generate a test case for each object type. IBM uses gray-box testing in this area to minimize the amount of redundancy in the test suite.

IBM also uses gray-box testing for audit. One component of the test suite generates all the possible audit records available in the system to ensure that each audit record contains the correct information in the proper field. Other test components verify that the audit macro was implemented correctly in a given number of interfaces. Additionally, the audit mechanism is activated for the DAC tests, and the audit trail is examined to ensure the proper audit event types and sub-types are generated.

For object reuse, IBM tests to ensure that associated spaces and indexes are initialized with zeros upon allocation. Tests also ensure that the auto extend portions of these objects are initialized. The team has determined that during on-site testing, source code must be examined to ensure that remaining storage objects (e.g., main storage, hardware registers, process structures) are cleared as designed (e.g., first reference on read).

For I&A, IBM tests to ensure that, in order to sign on, a user must have a valid user profile and the correct password for that profile. Additionally, tests exist to ensure that a user must have *USE authority to a User Profile to submit batch jobs on that user's behalf.

For the system architecture aspect of testing, IBM demonstrates the following:

- All administrator functions (commands and APIs) execute correctly when the user has the appropriate authorities
- System domain object types cannot be directly accessed from user state programs
- All UDSS interfaces validate parameters to ensure that a user cannot pass to an interface a system pointer to a system domain object when it is expecting a pointer to a user domain object
- The hardware storage write-protect mechanism works as designed and that UDUS programs cannot directly modify system domain spaces
- Blocked MI instructions are blocked from use by user-created programs
- Unblocked MI instructions perform the correct DAC checks and parameter validation checks
- User exits work as designed
- The system runs hardware diagnostics during IPL to ensure that the hardware is functioning as designed

In addition, IBM has test cases to ensure that program adoption works as designed.

The team has performed a test coverage assessment of IBM’s test suite and has found it to be acceptable. Additionally, the evaluation team has generated numerous team tests, which are intended to expose any obvious errors in the system configuration (e.g., DAC not set properly on commands or system files) or security mechanisms (e.g., being able to modify a system pointer arbitrarily).

The evaluation team performed 3 weeks of on-site testing at IBM’s Rochester, MN, site. Several bugs were found and corrected during this activity.

For this RAMP action, the VSAs performed and monitored 6 weeks of testing. The SA team performed an audit of the testing output.

Conclusion

AS/400 satisfies the C2 System Testing requirement.

8.4. Documentation

8.4.1. Security Features User’s Guide

Statement of Requirement

A single summary, chapter, or manual in user documentation shall describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another.

Assessment of Product Compliance

Appendix B titled “The User’s Responsibility for Security” of the *AS/400 Security-Basic* [5] along with *AS/400 Security - Reference* [19], *System Programmer’s Interface Reference* [6], *AS/400 Basic Backup/Recovery Guide* [4], *AS/400 Programming: Control Language Reference Common CL Information, SC41-0030-01* [15], and other chapters in the *AS/400 Security-Basic* [5] combined serve as the SFUG. The appendix is designed to inform all users about what they should do to protect the applicable security features of the system.

Section 1 of the appendix describes what a user profile is and its security relevance to the user and the commands that a user can use to change certain values contained in the user profile. Special authority (attribute of the user profile) is explained in chapter 6 of the *AS/400 Security-Basic* [5].

Section 2 of the appendix instructs the user about password management:

- How to protect their password
- How to change their password
- How to choose their password

This section also informs the user what to do if they forget their password (e.g., call their system administrator and inquire on local site policy).

Section 5 of the appendix instructs the user how to sign on to the AS/400 and the usage of multiple sessions. This section also instructs the user not to leave a active workstation unattended.

Section 7 of the appendix explains authority rights (private and public) to external objects (i.e. *FILE). Also explained are the access modes and how a user can designate and display the authority to a particular object. Object ownership, usage of program adoption, and creation and modification of authorization lists (grouping objects) is detailed in chapter 8 of the *AS/400 Security-Basic* [5] along with the *System Programmer's Interface Reference* [6]. The commands that are used to grant or revoke authority rights (public and private) are also detailed in the chapter 8 of the *AS/400 Security-Basic* [5].

Section 8 of the appendix instructs users not to grant anyone access to their user profile; this prevents batch jobs from being started with the user's authority.

Section 9 of the appendix explains how printed material should be handled with regard to security. This includes how to protect the file while on the output queue.

Sections 10 and 11 of the appendix instruct users how to protect new and existing objects and also how to obtain information about each of the objects that they own and have authority to.

The above references give the user a complete view of what must be done to use the security features provided in the system to protect resources.

For this RAMP action, the VSAs ensured that appropriate updates were made to the Security Features User's Guide to reflect all new and changed functions (as appropriate).

Conclusion

AS/400 satisfies the C2 Security Features User's Guide requirement.

8.4.2. Trusted Facility Manual

Statement of Requirement

A manual addressed to the ADP System Administrator shall present cautions about functions and privileges that should be controlled when running a secure facility. The procedures for examining and maintaining the audit files as well as the detailed audit record structure for each type of audit event shall be given.

Assessment of Product Compliance

IBM has written a manual titled *Security - Enabling for C2* [10] which along with Appendix D of the *AS/400 Security - Reference* [19], *AS/400 Security-Basic* [5], *AS/400 Licensed Programs and New Release Guide* [12], *AS/400 Basic Backup/Recovery Guide* [4], *System Programmer's Interface Reference* [6], *AS/400 Local Device Configuration* [32], *AS/400 Software Installation* [12], *AS/400 System Operation* [33], and the *AS/400 Advanced Backup/Recovery Guide* [3] form the Trusted Facility Manual.

The *AS/400 Security-Basic* [5], and *AS/400 Basic Backup/Recovery Guide* [4], along with the *AS/400 Licensed Programs and New Release Guide* [12] provide the system officer with the general security procedures that must be followed to establish and maintain a secure system. This information includes the following:

- Instructions for installing the system
- Instructions for creating user and group profiles
- Instructions for setting up ownership
- Instructions for setting up resource authorities
- Instructions and guidelines for auditing security relevant events
- Instructions and guidelines for physical security of the processor, workstations, tapes, disks, printers and printed output
- Instructions and guidelines for backup and restore of user data and system data

The *Security - Enabling for C2* [10] instructs the security officer what specifically must be done to establish and maintain a C2 level of security, which includes the following:

- Instructions for setting system values (see section 4.1.3.13 “System Values” on page 4-65 for a list of security relevant system values)
- Instructions for setting the security officer password
- Instructions for restricting the usage of certain commands, programs, and APIs These include commands and programs that are not allowed to be used on a C2 system
- Instructions for restricting the usage of special authorities (see section 5.2.6 “User Classes and Special Authorities” on page 5-4 for a list and definition of special authorities)
- Instructions for setting up initial auditing, restricting the audit commands and interfaces, and protection of the audit journal
- Instructions for restricting communication capabilities
- Instructions for object reuse of system controlled resources and the manual degaussing of magnetic tape storage and proper erasure of floppy diskettes

The *AS/400 Security-Basic* [5] also explains how to verify that each action the security officer does to establish a C2 level of security has completed successfully and achieved the expected results.

Also included in this set are instructions and guidelines for changing an existing operational system to a C2 level of security.

The *AS/400 Security-Basic* [5], along with the *AS/400 Basic Backup/Recovery Guide* [4], instruct the security officer how to restore the system from a backup (in case the current system is unusable).

The *AS/400 Security-Basic* [5] discusses general security planning and procedure for both the security administrator and the system operator. Detailed information with regard to the physical security implications coupled with cautions about workstations and peripherals attached to AS/400 are also discussed in the *AS/400 Security-Basic* [5].

For this RAMP action, the VSAs ensured that all appropriate updates to the TFM were made.

Conclusion

AS/400 satisfies the C2 Trusted Facilities Manual requirement.

8.4.3. Test Documentation

Statement of Requirement

The system developer shall provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing.

Assessment of Product Compliance

IBM has created a test plan [21], which discusses the company's philosophy and approach to ensuring that it meets the C2 requirement for security testing. In addition, IBM provides documentation that identifies test cases and provides a map to interfaces.

The test cases themselves are documented in preambles to the test case source code. In most instances it was possible to ascertain the test's purpose, description, and expected results from the preambles. In some cases, the test source code had to be examined. The team found that this level of test documentation was reasonable and acceptable.

For this RAMP action, the SA team reviewed and approved the test plan. The SA team also examined the results of the testing.

Conclusion

AS/400 satisfies the C2 Test Documentation requirement.

8.4.4. Design Documentation

Statement of Requirement

Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB. If the TCB is composed of distinct modules, the interfaces between these modules shall be described.

Assessment of Product Compliance

IBM has provided a set of design documents that describe the philosophy of protection: how the philosophy of protection is translated into the TCB, interfaces into and between the TCB modules, and how the TCB protects itself.

The *Philosophy of Protection* [14] contains descriptions of the subjects, objects, privileges, and overall protection mechanisms of AS/400. The *AS/400 Rating Maintence Phase - Process* [16] has broken AS/400 into functional component areas, and specific documentation is provided at the component level, such as *Discretionary Access Control Documentation* [9], *AS/400 Security - Reference* [19], *AS/400 User Domain/System State Programs DAC Policy* [22] and the *AS/400 Security-Basic* [5], which describe how the material discussed in the *Philosophy of Protection* [14] is incorporated into AS/400.

The *System Programmer's Interface Reference* [6] and the *AS/400 Machine Interface Functional Reference* [13] describe the interfaces to the TCB.

Since AS/400 consists of functional component areas, each component has a high level design document called a Component Overview Document (COD) or a SLIC Internal Design (VID), which describes the internal (design, control blocks, interfaces, and object references) workings of each functional area.

The prologues for each module in the component may be used to provide a more detailed view.

Design Change Requests (DCR) may also supplement the COD or VID if changes have been made. The COD or VID together with the DCR and prologue form the complete design documentation for the component.

The basic hardware functions are described in the *Amazon User Instruction Set Architecture* [29], *Amazon Virtual Environment Architecture* [30], *Amazon Operating Environment Architecture* [31] and *SPD I/O Bus Architecture* [18] documents and are supplemented by documentation for each of the attached peripherals.

Conclusion

AS/400 satisfies the C2 Design Documentation requirement.

8.5. Ratings Maintenance Phase (RAMP) Requirements

The vendor shall have in place procedures, mechanisms, tools, and personnel that comply with the following Rating Maintenance Phase (RAMP) program requirements for the Trusted Product Evaluation Program, as specified in *Rating Maintenance Phase Program Document*, NCSC-TG-013, Version 2, 1 March 1995. Note that the definitions in Chapter 3 of NCSC-TG-013 apply to all statements of RAMP requirements.

8.5.1. Configuration Items

Statement of Requirement

Configuration items shall be identified by the vendor in an NSA-approved RM-Plan and shall encompass:

1. The components or subsystems, including software source and object code, that comprise the Trusted Computing Base (TCB).
2. Any hardware and/or software features that are used to periodically validate the correct operation of the TCB in satisfaction of the System Integrity requirement.
3. The informal or formal model of the security policy (at the B1 evaluation class).
4. The Security Features User’s Guide (SFUG).
5. The Trusted Facility Manual (TFM).
6. The test plan, the test procedures that show how the security mechanisms were tested, and the expected results of the security mechanisms’ functional testing, and related test documentation.
7. The design documentation.
8. The RM-Plan.

Assessment of Product Compliance

The AS/400 Rating Maintenance Phase - Process [16] document (also referenced as the RM-Plan) includes the components comprising the TCB in section 3. The configuration items (CIs) are divided into deliverable and non-deliverable CIs. The SFUG and TFM are listed as deliverable CIs, while the RM Plan, the design documentation, and the test plan, test procedures, and expected results are listed as non-deliverable CIs.

Appendix B of the RM Plan lists the deliverable configuration items making up the TCB by type, including documentation, hardware, and software CIs. The features used to periodically validate the correct operation of the TCB are included in this list.

Conclusion

AS/400 satisfies the Configuration Items requirement.

8.5.2. RAMP Evidence

Statement of Requirement

For each change, RAMP Evidence shall include the following:

1. A description of the change.
2. The issues and conclusions of the Security Analysis.
3. Identification of the CIs affected.
4. The status of the changes to the CIs (e.g., being implemented, or completed).

Assessment of Product Compliance

The RM Plan includes a sample RMR in Appendix A. This appendix includes sample data indicating what information is included in RAMP evidence. Section 6, which describes the DCR and PTF processes, and section 5, which includes the security checklist, show that a description of each change, the issues and conclusions of security analysis, a list of the CIs affected, and the status of changes to these CIs are included in the RAMP evidence for each design change request or temporary fix.

Conclusion

AS/400 satisfies the RAMP Evidence requirement.

8.5.3. Ratings Maintenance Plan

Statement of Requirement

The RM-Plan shall include the following:

1. Identification of the VSA(s) and the RCO, including their corporate position.
2. The division of technical responsibilities among VSAs (if more than one).
3. The original date of approval of the RM-Plan and the dates of all approved changes.
4. The policies and procedures for Security Analysis.
5. The procedures for complying with applicable interpretations.
6. The policy for using emergency procedures for correcting errors and for incorporating these corrections in subsequent scheduled product releases.
7. A convincing argument to show that the described mechanisms, procedures, and tools are sufficient to address all changes to the product, including new features, bug fixes, and changes to satisfy Applicable Requirements.
8. The procedures for a VSA-performed RAMP Audit.
9. The procedures for RM-Plan maintenance.
10. A list of all CIs.
11. The rationale for the chosen granularity of CIs.

12. A description of the format of the RAMP Evidence.
13. All updates necessary to reflect corrective measures taken after a RAMP process failure (e.g., failure to follow, or error in following, the RM-Plan), if one has occurred.

Assessment of Product Compliance

The RM Plan is divided into twelve sections and two appendices. Section 1 provides background and revision history. Section 2 is an introduction, while section 3 defines the TCB and gives a rationale for the granularity of TCB configuration items (CIs). The CIs themselves are listed in Appendix B. Section 4 defines roles and responsibilities of the Responsible Corporate Officer, the Vendor Point of Contact, and the Vendor Security Analysts (VSAs). Section 6 defines and gives principles and procedures for security configuration control and shows that all changes to the product are included in the RAMP process, and section 7 discusses RAMP cycles. Section 8 details compliance with TCSEC interpretations, section 9 describes the preparation and presentation of RAMP evidence, and section 10 describes breakdowns or exceptions in the RAMP process. Section 11 discusses management of the RM Plan itself, while section 12 describes RAMP audits. Appendix A provides a sample RMR.

Conclusion

AS/400 satisfies the Ratings Maintenance Plan requirement.

8.5.4. Ratings Maintenance Report

Statement of Requirement

Each RMR shall include the following:

1. A summary identifying each change that has been made since the previous evaluated release of the RAMP Product.
2. A description of all security-relevant changes and the Security Analysis of those changes.
3. A description of how the RAMP Product meets the Applicable Requirements.
4. Identification of all tools used for generating CIs.
5. The internal procedures used for restoring the RAMP process if the RAMP Cycle covered by the RMR included a process failure. The description of the internal procedures must include:
   - The nature of the failure;
   - The Security Analysis conducted to establish corrective measures and verify product trust;
   - Establishment of the missing trail of evidence linking the evaluated product to the RAMP Product.
6. Results of the VSA-conducted RAMP Audit.

Assessment of Product Compliance

Section 2 of the RMR includes a summary of the changes made to the AS/400 since the previous RAMP cycle, including all relevant DCRs and PTRs. The RMR includes sections on compliance with the TCSEC, summary of the security test suite results, and corrective measures taken on RAMP failures. Section 6 of the RMR describes the nature of any process failures that occurred and how they were corrected. Section 4 of the RMR describes the results of the VSA conducted RAMP Audit. Appendix A of the RMR documents the DCRs that were security relevant. This table includes a short analysis of the nature of the changes.

Conclusion

AS/400 satisfies the Ratings Maintenance Report requirement.

8.5.5. Responsible Corporate Officer

Statement of Requirement

The Responsible Corporate Officer, or if the RCO has designated a VBPOC to act on behalf of the RCO, the VBPOC, shall:

1. Always be identified while the vendor is participating in RAMP and shall be responsible for the overall management of the vendor's RAMP effort.
2. Identify at least one VSA at all times while rating maintenance actions are underway.
3. Be responsible for submitting a proposed RM-Plan during the initial evaluation and shall obtain approval of the RM-Plan before entering the Formal Evaluation Phase.
4. Ensure that all subsequent changes to the RM-Plan, to reflect all changes made in the vendor's implementation of the ratings maintenance process, are submitted to the NSA for approval.
5. Sign the cover letter of the proposed RM-Plan.
6. Sign the cover letter of the RMR.
7. Ensure that any requested RAMP Audit is conducted promptly following the request.
8. Be responsible for submitting, as directed by the TPOC, copies of the following materials at least four weeks in advance of the scheduled RAMP TRB:
    - The RMR;
    - The NSA-approved RM-Plan;
    - The Updated Final Evaluation Report (FER);
    - The proposed product description for the EPL.

Assessment of Product Compliance

Section 4 of the RM Plan identifies the Responsible Corporate Officer (RCO) and the Vendor Point of Contact (VPOC), who also serves as the lead VSA. Though the VPOC, who is also the lead VSA, is directly responsible for much of what is listed in this requirement, the RCO remains accountable.

Conclusion

AS/400 satisfies the Responsible Corporate Officer requirement.

8.5.6. Security Analysis

Statement of Requirement

Security Analysis shall include the following:

1. Examining changes to the RAMP Product for security relevance, including analyzing the effects on the TCB.
2. Reviewing the design of approved changes.
3. Ensuring that the RAMP Product is adequately tested, including ensuring adequate test coverage through modification of the tests as necessary.
4. Ensuring that all documentation needed to show compliance with the Applicable Requirements, including design and user documentation, is updated consistently to reflect all changes to the TCB.

A change shall be considered to affect the TCB if it alters code or documentation within the identified TCB boundary, changes the TCB boundary, augments the TCB, or indirectly affects the function of TCB elements.

A change shall be considered security-relevant if it directly affects any mechanism implementing identified security policies (e.g., discretionary access control (DAC), object reuse, TCB isolation) or if it directly affects the maintenance of security data.

Security Analysis shall encompass cumulative effects involving all CI changes. (For example, two otherwise acceptable changes may conflict in terms of security because one assumes conditions that no longer hold, given the other change.) Security Analysis shall also consider the effects of interrelationships among the security features of the RAMP Product.

Assessment of Product Compliance

Section 5 of the RM Plan defines security analysis and documents the principles and procedures for security analysis followed by the AS/400 vendor team. The definition and principles given in the RM Plan ensures that security analysts recognize when a change affects the TCB, when a change is considered security relevant, and that cumulative changes and interrelationships among security features are considered in security analysis. The procedures used for security analysis include use of a security checklist. This security checklist is filled out by developers and forwarded to the security analyst as part of the change review process. Both design change requests and program temporary fixes require use of the security checklist as part of the documented security analysis procedure. The change review process includes documentation of security functional test procedure changes or additions to support each security-relevant change. Changes to documentation are also included. Section 5 of the RM Plan also details the education process used to prepare for security analysis.

Conclusion

AS/400 satisfies the Security Analysis requirement.

8.5.7. Vendor Security Analyst

Statement of Requirement

A Vendor Security Analyst shall:

1. Successfully complete the NSA training program for VSAs (i.e., the VSA Class).
2. Deliver the vendor’s Quarterly Status Reports (QSRs) to the vendor forum on the required schedule.

3. Conduct, supervise, or monitor all Security Analysis tasks according to the approved RM-Plan.
4. Review the Security Analysis prior to the submission of the RMR for the rating maintenance action.
5. Conduct an initial RAMP Audit prior to the original evaluation team's testing of the TCB. The results of this initial RAMP Audit must be provided to the evaluation team.
6. Conduct at least one RAMP Audit[1] for each RAMP Cycle. The results of the RAMP Audit must be included in the next quarterly status report following the RAMP Audit.
7. Ensure that before the RMR is submitted, the relevant parts[2] of the entire security functional test suite used in the original evaluation, as updated during the RAMP Cycle, are successfully executed on a representative sample of hardware.
8. Demonstrate to the TRB that Security Analysis has been conducted according to the approved RM-Plan in that RAMP Cycle.

Assessment of Product Compliance

The RM Plan states that the AS/400 Division will maintain at least one Vendor Security Analyst (VSA). At the present time, the AS/400 Division has multiple VSAs who have completed the required NSA training program and who perform and check security analysis for the identified configuration items making up the AS/400 product. These VSAs, who are all located in Rochester, MN, participate in the change control boards for hardware, firmware, and software changes. The TPOC and the SA team have observed several change control board meetings at the invitation of the VSAs participating. The RMR, as documented in the RM Plan, is prepared based on the materials prepared and reviewed as part of the change review cycle.

As documented in the RM Plan, the lead AS/400 VSA serves as the interface to NSA, coordinates the activities of these VSAs, and also coordinates the activities of the other security analysts and the C2 product test team leader. The lead VSA posts the Quarterly Status Report to the vendor forums as required, coordinates the vendor RAMP audit for each RAMP cycle, and coordinates review of the results of security functional testing. The AS/400 lead VSA has control over AS/400 products developed at other sites with respect to changes that affect security. The lead VSA also coordinates presentations by VSAs, the TPOC, and SA Team members for Future Change Review Boards (FCRBs) and TRBs.

Conclusion

AS/400 satisfies the Vendor Security Analyst requirement

8.6. Interpreted TCSEC Requirements

As the AS/400 previous RAMP action was in April, 1997, all interpretations listed in the "Interpreted TCSEC Requirements" potentially apply to AS/400. All interpretations dated prior to May 1994 were covered in the original evaluation. There are no new interpretations published since the last Final Evaluation Report (FER) and RAMP action.

[1] This audit can be conducted in conjunction with the TPOC's audit.

[2] In general, the entire test suite must be executed for each RAMP action because it is not feasible to determine with confidence which tests could not have been affected by the changes. If, however, the changes are limited in scope, or there are parts of the test suite that can be shown to be unaffected, a subset of the tests may be performed. The rationale for any such limitations must be presented to the RAMP TRB Panel.

Table 8-1. Interpreted TCSEC Requirements

Interpretation	Level
I-0001 “Delayed enforcement of authorization change” The AS/400 documents the immediate revocation procedure in the *AS/400 Security - Enabling for C2* [10].	C1, C2, ...
I-0002 “Delayed revocation of DAC access” AS/400 is in compliance with this interpretation.	C1, C2, ...
I-0003 “Access validation after object label change” NOT APPLICABLE	B1, B2, ...
I-0004 “Enforcement of Audit settings consistent with protection goals” The AS/400 documents the procedure for immediate enforcement of audit setting changes in the *AS/400 Security - Enabling for C2* [10].	C2, B1, ...
I-0005 “Action for audit log overflow” The AS/400 provides the Audit End Action (QAUDENDACN) system value which allows an administrator to specify the action to take if the system in unable to log records to the audit journal. This is documented in the *AS/400 Security - Reference* [19], which is part of the TFM.	C2, B1, ...
I-0006 “Audit of user-id for invalid login” The AS/400 records the character string supplied as the user identity in the Password (PW) audit entry when a login failure occurs.	C2, B1, ...
I-0007 “Assigning device level range” NOT APPLICABLE	B2, B3, ...
I-0020 “DAC authority for assignment” The *AS/400 CL Reference* [15], which is part of the SFUG, documents who can grant, revoke, or edit object authorities.	C1, C2, ...
I-0022 “One set of banner pages around multiple outputs” NOT APPLICABLE	B1, B2, ...
I-0039 “Multilevel printers and page labeling” NOT APPLICABLE	B1, B2, ...
I-0040 “Requirements for overwrite label capability” NOT APPLICABLE	B1, B2, ...
I-0041 “Object reuse applies to all system resources”	C2, B1, ...
I-0043 “Auditing use of unnamed pipe” The AS/400 does not support unnamed pipes.	C2, B1, ...

Table 8-1. Interpreted TCSEC Requirements

Interpretation	Level
I-0046 “Detailed audit record structure” The AS/400 description of the audit record structure is contained in the Security Reference Manual [19], part of the TFM.	C2, B1, ...
I-0053 “Public objects and DAC” All external object types in AS/400 receive the same DAC protection. Public authority *USE (or READ) does not give object change, create, or delete authority.	C1, B1, ...
I-0069 “Flexibility in packaging TFM” A hardcopy of the manuals that comprise the TFM are orderable and available from IBM.	C1, C2, ...
I-0073 “OK to audit decision regardless of whether action completed” AS/400 is in compliance with this interpretation.	C2, B1, ...
I-0084 “Audit least disruptive action” NOT APPLICABLE	B3, A1
I-0096 “Blanking passwords” The password entry fields on workstations are non-display (don’t show entered value). The password data is stored in an encypted form and is not decrypted (not made available in clear text form) to any user of the system.	C1, C2, ...
I-0144 “Availability of diagnostics” The diagnostic functions and features are available and exercisable by the end-user on an “on-demand” basis.	C1, C2, ...
I-0147 “Administrator role may include operator functions” (See section 9.2 “Trusted Facility Management” on page 9-2)	B2
I-0170 “Functional tests required for object reuse” The TCB interfaces for dumping the internal contents of jobs and objects require that explicit authority to the interfaces be granted by an administrator.	C2, B1, ...
I-0172 “Audit of imminent security violations” NOT APPLICABLE	B3, A1
I-0192 “Interface manuals as design documentation” See section 8.4.4 “Design Documentation” on page 8-10.	C1, C2, ...
I-0193 “Standard system books as design documentation” See section 8.4.4 “Design Documentation” on page 8-10.	C1, C2, ...

Table 8-1. Interpreted TCSEC Requirements

Interpretation	Level
I-0213 “Administrator interface is part of TCB” As described in the AS/400 Test Plan: Administrator functions In compliance with TCSEC Interpretation I-0213, a “functional” test is provided for every command and API that is identified to be an administrator function. The functional test shows that, with the proper authority, the function is executed correctly. An administrator function is any command or API which either: – requires any user profile special authority (i.e *ALLOBJ, *JOBCTL, *SECADM, etc.). OR – is executed in the TCB environment and is shipped with public *EXCLUDE authority so that only a system administrator can use it. A list of administrator functions is maintained in the Security - Enabling for C2 [10] book, which is part of the TFM.	C2, B1, ...
I-0222 “Passwords not acceptable for DAC” The AS/400 does not use passwords for DAC. Passwords are used only for authentication.	C1, C2, ...
I-0234 “One-time authentication mechanisms can be acceptable” While allowed, one-time authentication mechanisms are not implemented.	C1, C2, ...
I-0239 “Subject access revocation after change in user clearance” NOT APPLICABLE	B1, B2, ...
I-0240 “Passwords may be used for card input” The AS/400 does not support card input.	C1, C2, ...
I-0244 “Flexibility in packaging SFUG” A hardcopy of the manuals that comprise the SFUG are orderable and available from IBM.	C1, C2, ...
I-0253 “Default page marking format” NOT APPLICABLE	B1, B2, ...
I-0254 “UNIX-style manual pages as DTLS” NOT APPLICABLE	B2, B3, ...
I-0275 “Single-level printers and page labeling” NOT APPLICABLE	B1, B2, ...

Table 8-1. Interpreted TCSEC Requirements

Interpretation	Level
I-0281 “Testing System Architecture functions” Tests are included in the test suite that explicitly exercise and verify the System Architecture protection mechanisms.	C1, C2, ...
I-0285 “CM comparison source or object?” NOT APPLICABLE	B2, B3, ...
I-0286 “Auditing unadvertised TCB interfaces” AS/400 is in compliance with this interpretation.	C2, B1, ...
I-0288 “Actions allowed before I&A” AS/400 is in compliance with this interpretation.	C1, C2, ...
I-0302 “Trusted path required for all authentication” NOT APPLICABLE	B2, B3, ...
I-0312 “Set-ID and the DAC requirement” AS/400 is in compliance with this interpretation.	C1, C2, ...
I-0314 “Password changes do not require authentication” In accordance with good practice, AS/400 requires re-authentication for a user to change their password.	C1, C2, ...
I-0315 “Restricted to trusted administrative interface” NOT APPLICABLE	B3, A1

Section 9
Features and Assurance Beyond The Evaluated Rating

⚠

Additional features do not provide the additional assurance of the higher rating.

9.1. Discretionary Access Control

B3 Requirement

The TCB shall define and control access between named users and named objects (e.g., files and programs) in the ADP system. The enforcement mechanism (e.g., access control lists) shall allow users to specify and control sharing of those objects, and shall provide controls to limit propagation of access rights. The discretionary access control mechanism shall, either by explicit user action or by default, provide that objects are protected from unauthorized access. These access controls shall be capable of specifying, for each named object, a list of named individuals and a list of groups of named individuals with their respective modes of access to that object. Furthermore, for each such named object, it shall be possible to specify a list of named individuals and a list of groups of named individuals for which no access to the object is to be given. Access permission to an object by users not already possessing access permission shall only be assigned by authorized users.

Applicable Features

AS/400 enforces a Discretionary Access Control (DAC) policy, as described in section 6.2 “Discretionary Access Control” on page 6-4. The DAC mechanism allows named users to define and control access to named objects through the use of authorities. Named users are those users with user or group profiles, as described in section 5.2 “Subjects” on page 5-2. The DAC policy is to perform DAC checks as part of each MI instruction that accesses an object with a system pointer, as discussed in section 3.2.2 “Machine Interface” on page 3-39. Additional checks are performed as part of processing CL commands and APIs.

AS/400 defines and controls access between subjects and named objects. (See section 4.1.1.1 “External Objects” on page 4-2.) The DAC enforcement mechanisms, authorities, authorization lists, and special authorities, allow a user to specify and control sharing of protected objects between named individuals, groups, or both. External objects are protected by default; when they are created, only the owner has authority to the object. Trusted users with appropriate special authority also have access to the object. AS/400 provides public authority (*PUBLIC). The *PUBLIC authority default in the C2 system is *EXCLUDE.

The group profile mechanism, described in section 5.2 “Subjects” on page 5-2, allows users to belong to groups.

Controls on the propagation of authority for named objects are discussed in section 6.2.7 “Propagation of Authority” on page 6-15.

Authority and authorization list mechanisms provide the capability to include or exclude access to the granularity of a single user. Access permission to an object by users not already possessing access permission can be granted only by a user with the appropriate authority (object owner, user with *OBJMGT or *AUTLMGT authority).

The Authorization List mechanism provides the capability to group a set of named objects on a list. User and Group profiles may point to the authorization list with their respective authority to all objects on the list.

Conclusion

AS/400 satisfies the B3 Discretionary Access Control requirement.

9.2. Trusted Facility Management

B2 Requirement

The TCB shall support separate operator and administrator functions.

Applicable Features

Through the use of special authorities, AS/400 allows sites to define roles, such as administrator, operator, and auditor. Special authorities are used to specify the type of actions a user can perform on system resources. A user can be given none or more special authorities. These special authorities, described in section 5.2.6 “User Classes and Special Authorities” on page 5-4, are: All Object (*ALLOBJ), Security Administrator (*SECADM), Job Control (*JOBCTL), Spool Control (*SPLCTL), Save System (*SAVSYS), Service (*SERVICE), and Audit (*AUDIT).

The TFM provides the list of commands that are considered operator commands. These commands require *JOBCTL, *SPLCTL, and *SAVSYS special authorities. Another list of commands are considered administrator commands, and these commands require the *SECADM special authority. There are no relationships between special authorities (i.e., incomparable). Thus, a user with *JOBCTL, *SPLCTL, and *SAVSYS cannot execute the administrator commands.

Conclusion

AS/400 satisfies the B2 Trusted Facility Management requirement and interpretation I-0147.

Section 10
Evaluator Comments

Audit

To support the audit requirement of the TCSEC, AS/400 has the capability to audit most user activities at the TCB Interface level. This level of granularity results in large amounts of audit data being produced for a single user action. Although AS/400 provides good tools for both preselection and postselection of audit events, the level of granularity makes it difficult to determine easily what actions have taken place on the system.

Cryptographic Support

The evaluation team performed no analysis of the cryptographic support provided by AS/400 and can pass no judgment on the algorithm it uses. The team ensured that the cryptographic module interfaces are subject to the DAC policy and that no residual data can be obtained from the buffers that are used.

Appendix A
Evaluated Hardware

The IBM AS/400 system that has been evaluated comes in two generations, Version 3 and Version 4. The version 3 systems come in three machine types, 9401, 9402, and 9406, which have four different types of enclosures. The 9401 machine type enclosure is a standard workstation type while the 9402 enclosure is a 2-meter tower. The 9406 has two different enclosure types, a 2-meter tower and a 3 meter tower.

The version 4 systems come in two machine types: 9401 and 9406. The 9401 machine type enclosure is a standard workstation type while the 9406 may have either single or double, 1- or 2-meter towers depending on the features, I/O, DASD, and memory.

These enclosures contain all elements of the AS/400 except workstations and printers. The precise machine and model numbers making up the Trusted Computing Base (TCB) are specified below. The models range from the 150 with a single processor and no cache to the 12 way multiprocessor 650 feature code 2243 with 4 MB of L2 cache for each processor. The following models of AS/400 make up the evaluated configuration:

Table A-1. Basic System Configuration Models

Machine type	Model	Features	Processor Type
9401	150	2269 2270	A10
9402	400	2130 2131 2132 2133	A10
9402	40S	2109 2110 2111 2112	A10
9406	500	2140 2141 2142	A10
9406	510	2143 2144	A10
9406	50S	2120 2121 2122	A10
9406	600	2129 2134 2135 2136	A10
9406	S10	2118 2119	A10
9406	620	2175 2179 2180 2181 2182	A35
9406	S20	2161 2163 2165 2166 2177 2178	A35

Table A-2. Complex System Configuration Models

Machine type	Model	Features	Processor Type
9406	53S	2154 2155 2156 2157	A30
9406	530	2150 2151 2152 2153 2162	A30
9406	640	2237 2238 2239	A35
9406	S30	2257 2258 2259 2260 2320 2321 2322	A35

Table A-2. Complex System Configuration Models

Machine type	Model	Features	Processor Type
9406	650	2240 2243	A35
9406	S40	2256 2261	A35
9406	SB1	2310 2311	A35

A set of peripheral devices consisting of workstations, DASD, diskette drives, magnetic tapes, and printers may be attached to the AS/400 as needed. These peripherals are all directly attached to the IOPs on the evaluated system by twisted pair or twinax cables, and all interface to the IOPs are by protocols described previously in the IOP section above. The tapes, DASD, and diskettes are housed within the same enclosures as the CPU, while the printers and workstations may be outside of the computer room on multidrop lines. The peripheral devices in the TCB include the following:

- The following DASD device types:
  - 6105* 6107* 6109 6600 6601 6602 6603 6605 6606 6607 9336* 9337 6713

    * = Version 3 hardware only
- The following diskette device types:
  - 6131 6132 6133 9331
- The following magnetic tape device types:
  - 2440 3422 3430 3480 3490 6341 6342 6343 6344 6346 6347 6348 6349 6366 6368 6369 6378 6379 6380 6390 7208 9346 9347 9348 3570 3590 6335
- The following optical device types:
  - 3995 632A 6320 6321
- The following display station device types:
  - 3179 3180 3196 3197 3277 3278 3279 3476 3477 3486 3487 5150 5251 5291 5292 3101 3151 3161 3162 3163 3164 4247 6404 6408 6414
- The following printer device types:
  - 4224 4245 5224 5225 5256 5219 5583 3287 4214 3812 5262 6252 4234 4019 4201 4202 4207 4208 4216 5204
- The 4755 Microchannel Cryptographic Adapter Feature 2620 and 2628

Appendix B
Evaluated Software

The AS/400 software consists of the following products: SLIC, OS/400, Cryptographic Services, SQL/400, Query/400, the Application Development Tools Utility (SEU), and the Advanced Function Printing Fonts/400. With the exception of Advanced Function Printing Fonts/400 (which consists only of bitmaps of fonts), each product is made up of one or more components. The code in some components is not executed in the evaluated configuration because users are prevented by Discretionary Access Control (DAC) from using the interfaces to this code. All components containing code that can be executed in the evaluated configuration are identified in the tables that follow.

B.1. Ordering Information

When ordering AS/400 V4R1M0, users must specify that they want Feature Code 1920. This feature code is required to meet the C2 level of security. Feature Code 1920 consists of a special Cumulative Program Temporary Fix (CUM) CD-ROM of evaluated PTFs and special programs that, when used with the AS/400 V4R1M0 Licensed Internal Code, Operating System/400, and certain Licensed Products, provides the basis for a C2 system. The Feature code 1920 CUM CD-ROM includes a set of programs to assist with the customizing of your system to satisfy C2 security requirements. These programs are packaged as a PTF on the CD-ROM and are loaded into the QSYS library as part of the installation process. Instructions for running these programs are contained in the manual Security - Enabling for C2. The Feature Code 1920 CUM package replaces the normal CUM PTF package, which should not be loaded. A complete order for installation of a C2 system has a CD-ROM containing the Licensed Internal Code and a CD-ROM containing the Operating System/400 and, stacked on the same CD-ROM, the optionally installable and optionally purchasable evaluated Licensed Products: Advanced Printing Fonts (5716-FNT), Source Entry Utility (SEU) (5716-PW1), Query/400 (5769-QU1), and SAA Structured Query Language/400 (5769-ST1). If the Common Cryptographic Architecture Services/400 (5799-FRF) PRPQ was ordered, it is on a separate CD-ROM. Feature Code 1920 PTFs with the C2 customizing programs are also on a separate CD-ROM.

SLIC CONFIGURATION ITEMS

The SLIC Configuration Items (CIs) are composites of related components. These components and the parts created within them are tracked within the Configuration Management System.

- Configuration Item (CI) - Functional Grouping
- TCB Group Status within the TCB
  - N = Not in TCB
  - Y = Included in TCB
  - F = Specific Functions may be included in TCB
- Components - SLIC Components included in the group

Table B-1. SLIC Configuration Items

Configuration Item	TCB	SLIC Components
AS/36	N	NU
BUILD	N	NDP
COMMON FUNCTIONS	Y	CF MC RC RT RTTB SO NM CR C3 MS ND NR PA RA BSF SI SS TEX TS DLUX T1 AM CT HPR LC LM LNKT PPP P0Y SC SK TB TE TI TL TRAN TQ TT TU TX TZ YI AS LO OT PI PS SB T2 F2 NTB SNA CP
COMMUNICATIONS	N	MPAI MPTN SNAP ZSP DS TW ZSG TG TO AB YYCT
DASD STORAGE MGMNT	Y	SMD PUMP
DATA BASE	Y	II IX DBDS DBCR DBFR DBIX DBIP DBLD DBMN DBMP DBOC DBRT DBSL CO SORT JO JOIP JORT LD
FSIOP/AIX	N	FPB FPF FPH FPN (FSIOP) X6 XC X6T X6V (AIX IOP)
IDE	Y	IDE LID RSA AMES HW1 HW2 HW3 HW4
INTEGRATED FILE SYS	Y	P0D(DIR) P0F(PFS) P0N(NFS) ZER BS FS(FILE SERVER) TLI (TRANSPORT LAYER IF) ZLS (V4R15 NATIVE SMB SERVER)
INTEGRITY VERIFICATION	Y	BATS[a]
I/O(NON COMM)	Y	TA POWR EL ELA EREP RISP DD IO IOAT IOCM IOCT IODK IODS IOLI IOPI IOMI IOP IOPR IOPT IOST IOSV MO EA
PERF	N	PDC (PD/PT)
PROGRAM MODEL	Y	AI CC DO DI BN MM M2 PG OX XC XL MX PR XOM(SOM/400)
RESOURCE MANAGEMENT	Y	EM PM QM RM RMAC RMAG RMIS RMPR RMSC RMSL RMTM PX PXSP(SYNCH) PXSG(SIGNALS) MA MASO (THREADS)AU MN EX QS PXPM (PROC MGR)
SERVICE	F	DM INOS INUC IT LL MAC ML RI RIBM RIDA RIEM RIFL RIRS RITR RIVL SD SDDI S3(DST SF SFCC SFDT SFDV SFFW VL VT VTAT VTVR SFHS VTSI (PIRANHA)
STORAGE MANAGEMENT	Y	SM SMA SMM HM(MWS)
XPORT & IPCF	F	BM(XPORT MGR) IP(IPCF)

a. While the BATS are referenced as a single item, they are physically an integral part of several SLIC components distributed through the layer. They are treated as a single entity during their execution.

OS/400 CONFIGURATION ITEMS

These Configuration Items (CIs) are composites of related components. These components and the parts created within them are tracked within the Configuration Management System.

- Configuration Item (CI) - Functional Grouping
- TCB Group Status within the TCB
  - N = Not in TCB
  - Y = Included in TCB
  - F = Specific Functions may be included in TCB
- Components - OS/400 Components included in the group

Table B-2. OS/400 Configuration Items

CONFIGURATION ITEM	TCB	COMPONENTS
AS/36-38	N	CI EX MU NU OR UT Y1
Communications	N	A1 AC AL AP BI BS CM CM0 CU DX IC IF L6 LC LM LNKT LS ND NF NM NPS NX OK OL ON PA S2 SA SI SL SOC TOA TOC TOD TOI TOM TON TOR TOS Y2 YF YYCT ZD ZPAI ZSP
Data Base	Y	CN DB ID JO SP TN TS XOB
Industry Support	N	FN LY RF
Installation Mgmt	Y	ES IA MA PD PZ RU SCA SCB SM SX SZ TI TT UP XNA
IWS Support	N	YUN
I/O	Y	AR CJ CO DK DR EM FS GD GS IMG MO PI PK PQ RA SF SK SN SO SW T1 T3 TA TV UC WP WPEE WS ZBS ZXC
Languages Support	F	BN EF FO LE MZ PR RE REZ RM XOT ZRU
MRI/Pubs/NLS	F	BMG BMR KJ LG TB WH ZP
Office	N	FI OC OD OE OF OG OH OJ OP OQ OS OT ZMF
Posix/IFS	Y	P0D P0F P0L P0Q P0S P0W
Query	F	QM QQ QU QX RW SQ X1
Server	N	FPAD FPF FPH FPI FPN FPP FPZ HC HE HNA NTB P0A P0N PWFS X6A X6S ZCA ZCP

Table B-2. OS/400 Configuration Items

CONFIGURATION ITEM	TCB	COMPONENTS
Support Services	F	5B AE BR C3 CC D0 DD DF EA EC ED EG EZ FV IZA KA LZA MN NT P0Z PM PT QA ST SVR TE TES TQ TW UH UR YPE ZCC
System Nucleus	Y	CA CD CL CP DC DM DV HF IN LI LP MH MHQ OX P0MS RC RZ SC SEC SH SR SY UI US WC WD WT WV XIC
Utilities	F	DZ FPD PY SU ZDM

Table B-3. Product 5799-FRF, Cryptographic Services

ID	Short Name	Full Name
C3	Crypto	Cryptographic Services (Aegis)
C4	Security API	Security API, Server, Key Storage

Table B-4. Product 5769-ST1, DB2 Query Manager and SQL Development Kit for AS/400

ID	Short Name	Full Name
QM	Query Manager	SQL/400 Query Manager
SQ	SQL COMP	SQL PRODUCT COMPONENT
S1	ISQL	SQL/400 Interactive

Table B-5. Product 5769-QU1, Query for AS/400

ID	Short Name	Full Name
QU	QRY/400	DfnTime Query/400: Definition-time function

Table B-6. Product 5716-PW1, Application Development ToolSet for AS/400

ID	Short Name	Full Name
SU	SEU	SOURCE ENTRY UTILITY

Table B-7. Product 5716-FNT, Advanced Function Printing Fonts for AS/400

ID	Short Name	Full Name
N/A	N/A	N/A

Appendix C
Draft Product Bulletin/EPL Entry

Report No: CSC-EPL-95/006.C

AS OF: TBD

MAINTAINED PRODUCT:	AS/400 with OS/400 Version 4 Release 1 Modification (0) (V4R1M0) with Feature Code 1920 applied running on RISC based Hardware
PREVIOUS PRODUCTS:	AS/400 with OS/400 V2R3M0 AS/400 with OS/400 V3R0M5 AS/400 with OS/400 V3R2M0

VENDOR: International Business Machines Corporation (IBM)

EVALUATION CLASS: C2

PRODUCT DESCRIPTION:

The IBM AS/400 is a fully integrated computing system that includes a fully relational data base with a Structured Query Language (SQL) interface, a Personal Computer (PC) and Portable Operating System Interface (POSIX) compatible hierarchical file system, a Command Language that provides a consistent command interface, as well as compilable procedures, an editor, and a full range of operating system functions, including a consistent security model across the system. The AS/400 evaluated configuration provides a multiprocessing environment scalable across a range from a few users to thousands of users.

The AS/400 is an object-based system with strong object typing and dynamic binding. Its addressing is based on the single level store model, providing a single 64-bit (16 M terabyte) address space that all users share. All data is contained in encapsulated objects managed by the operating system. Another key feature of AS/400 is the concept of an abstract interface, known as the Machine Interface (MI). A trusted translator translates the MI code to Reduced Instruction Set Computer (RISC) instructions that are executable on the hardware. The translator also converts 48-bit applications from Internal MicroProgram Instruction (IMPI) releases to provide full 64-bit capability.

Since all sharable data is contained in encapsulated objects, discretionary access control (DAC) is maintained by each object manager using a system-wide, common function to verify authorization of any user to any instantiation of an object. The AS/400 DAC mechanism provides authorities on both a private (individual or groups) and public (any authorized user) basis. A user may obtain authority from any one of up to 16 groups. In addition, the capability to adopt (gain) authority from a program owner is provided. Two levels (system groupings plus individual data and object authorities) of DAC granularity are provided to suit varying user needs.

Object reuse on the AS/400 is controlled primarily by storage management when storage objects are allocated. When objects are reused, the object manager either reuses them on a user basis or clears all residual data.

AS/400 provides accountability via a centralized identification and authentication mechanism in the operating system. All users are registered via a user profile object created by a trusted administrator. Associated with each user profile is a one-way Data Encryption Standard (DES) encrypted password. The AS/400 provides numerous options as to the makeup and control of passwords, allowing specific installations to meet their local requirements. In addition, AS/400 provides uid and gid support for user and group profiles.

A highly configurable set of auditing capabilities is available on the AS/400. These auditing capabilities fall into three categories: actions, objects, and users. AS/400 provides the ability to selectively audit very specific preselection by narrowing the scope of the audit to only the desired actions, objects, or users, thereby reducing the system resources needed for auditing. In addition, ease of postselection is provided via the SQL interface.

AS/400 provides data integrity features in addition to its security features. AS/400 assures system isolation by explicitly verifying all system pointers and space pointers passed into its Trusted Computing Base (TCB) via application programming interfaces, kernel system calls, commands, and MI instructions. In addition, the concepts of user state and system state are implemented in hardware, with full software support. The hardware provides a storage protection feature that restricts user state programs to read only or no access at all to protected memory locations. The encapsulation of programs and strong typing prevents data from being executed as a program. Since only program objects can be executed, and only within an operating system controlled process, all interprocess communications are controlled by the AS/400 and are auditable. The use of logical files (i.e., views) in the database provides the user a mechanism to specify access control to the record/column/element level of a physical file. (Note: authorization lists and private authorities are applied at the file level, not at the record/column/element level.) The SQL grant commands, as well as the referential integrity and triggers capability, use the same DAC mechanism as the rest of the system.

The evaluated configuration supports peripherals including: tape, direct access storage device (DASD), diskette, workstations, and optical devices. The software is distributed by CD-ROM.

PRODUCT STATUS:

The V4R1 version of OS/400 is currently orderable and the RISC-based e-systems and e-servers are currently being manufactured. The V4R1 version of OS/400 supersedes the original evaluated version, V2R3, and the first version of AS/400 to complete the RAMP program, V3R0.5, which have been withdrawn from the market. The most recent version of AS/400 to complete the Rating Maintenance Phase (RAMP) program, V3R2, remains available for IMPI-based machines. For ordering information, call IBM at 1-800-426-3333 or send e-mail to askibm@info.ibm.com.

The Vendor Point of Contact for the IBM AS/400 is Rick Sanders (507-253-4197 or rrsander@us.ibm.com), who can be contacted with questions relative to the evaluation.

SECURITY EVALUATION SUMMARY:

The security protection provided by the AS/400 Version 2 Release 3 had been initially evaluated by the National Security Agency to meet C2 in October of 1995.

The AS/400 V3R0.5 evaluated configuration was taken through the National Security Agency's RAMP program and evaluated as meeting C2 in September of 1996. This RAMP included the Advanced Series hardware.

The security protection provided by the AS/400 Version 3 Release 2 Modification 0 with Feature Code 1920 has been taken through the RAMP program to ensure that AS/400 continued to meet the “DoD Trusted System Evaluation Criteria” and relevant interpretations. The V3R2 version was awarded C2 in October of 1997.

The V4R1 system, which includes a port from IMPI to RISC hardware that required a significant rewrite of the Licensed Internal Code (LIC), went through both the RAMP program and was evaluated by a Security Analysis (SA) team. The conclusion is the IBM AS/400, running V4R1M0 on IBM RISC-based hardware as defined in the Trusted Facility Manual (TFM), satisfies all specified requirements of the Criteria for class C2.

The changes in the AS/400 TCB in V4R1M0 included support for optical devices and several new external object types. For a complete description of these changes and how the AS/400 V4R1 satisfies each requirement of the TCSEC, see the Final Evaluation Report, International Business Machines Corporation, Application System/400 (Report CSC-FER-95/006.C).

In addition to the C2 requirements, IBM AS/400 satisfies the class B3 requirement for Discretionary Access Control, and the class B2 requirement for Trusted Facility Management. Credit given above the overall rating recognizes mechanisms implemented beyond those strictly required within the assigned rating class. Security testing was not conducted above the rated level, and the reader should not infer any assurance above this level.

IBM plans to continue to participate in the TPEP process for AS/400.

ENVIRONMENTAL STRENGTHS:

The IBM AS/400 is an object-based system with a fully integrated relational data base with an SQL interface. Since AS/400 implements the concept of an abstract interface, known as the Machine Interface (MI), AS/400 applications can be ported to another AS/400 using a completely different hardware platform without recompiling the applications, as shown in the current port to RISC.

Appendix D
Relevant Decisions

1. No decisions regarding AS/400 have been published since the original evaluation.

Appendix E
Acronyms and Nomenclature

A

ACB	Activation Control Block
AG	Activation Group
AI	Activation and Invocation
AMCB	Application Main Control Block
ANSI	American National Standards Institute
APAR	Authorized Program Analysis Report
API	Application Program Interface
ASCII	American National Standard Code for Information Interchange
ASM	Auxiliary Storage Management
ASP	Auxiliary Storage Pool
AUT	Authorized Users Table

B

BCA	Bus Control Adapter
BCU	Bus Control Unit
BTM	Bus Transport Mechanism
BUM	Bus Unit Message

C

CA	Command Analyzer
CCSID	Coded Character Set Identifier
CDMF	Commercial Data Masking Facility
CL	Command Language
CMVC	Configuration Management Version Control
COBOL	Common Business Oriented Language
COD	Component Overview Document
CPP	Command Processing Program
CPU	Central Processing Unit
CS	Cursor Stability
CSEH	Component Specific Exception Handler

D

DAC	Discretionary Access Control
DASD	Direct Access Storage Device
DBMS	Data Base Management System
DCG	Design Control Group
DCR	Design Change Request
DDS	Data Description Specification
DES	Data Encryption Standard
DIA	Document Interchange Architecture
DMA	Direct Memory Access
DMCQ	Data Management Communication Queue
DoD	Department of Defense
DR	Data Relocate
DRAM	Dynamic Random Access Memory
DST	Dedicated Service Tools

E

EBCDIC	Extended Binary Coded Decimal Interchange Code
ECC	Error Correction Code
EMD	Evaluation Milestone Decisions
EPA	Encapsulated Program Architecture
EPL	Evaluated Products List
EPT	Entry Point Table

F

FACB	File Availability Control Block
FER	Final Evaluation Report
FIFO	First-In-First-Out
FM	Function Manager

G

GID	Group Identification Number

H

HLL	High Level Language
HPT	Hardware Page Table
HRI	Hardware Resource Information
HSP	Hardware Storage Protection
HW	Hardware

I

IA	Identification and Authentication
IBM	International Business Machines
ICB	Invocation Control Block
IDSS	Integrated Development Support System
IEEE	Institute of Electrical and Electronics Engineers
IIS	I/O Instruction Stack
ILE	Integrated Language Environment
IMPI	Internal MicroProgram Instruction
INFCB	Inline File Control Block

IO	Input/Output
IOA	I/O Adapter
IOBU	I/O Bus Unit
IOP	Input/Output Processor
IPAR	Initial Product Assessment Report
IPC	Interprocess Communication
IPCF	Interprocess communication facility
IPL	Initial Program Load
IPTR	Intensive Preliminary Technical Review
IR	Instruction Relocate
IRP	Intermediate Representation of a Program
ISDN	Integrated Services Digital Network
ISF	Invocation Stack Frame
IWA	Invocation Work Area

J

JMQ	Job Message Queue
JOBD	Job Description
JOBQ	Job Queue

K

K	Kilobyte
KB	Kilobyte
KMRT	Kernel Mode Routing Table

L

LDA	Local Data Area
LEAR	Lock Exclusive Allow Read
LENR	Lock Exclusive No Read
LFS	Logical File System
LIFO	Last-In-Last-Out
LP	Licensed Product
LSRD	Lock Shared Read
LSRO	Lock Shared Read Only
LSUP	Lock Shared Update
LU	Language and Utility
LUD	Logical Unit Description
LWPCB	Light Weight Process Control Block

M

MB	Megabyte
MBV	Module Binding Vector
MESI	Modify, Exclusive, Share, Invalid
MFRR	Most Favored Request Register
MI	Machine Interface
MIRQ	MI Response Queue
MS	Main Store
MSCP	Machine Service Control Point
MSM	Main Storage Management
MSR	Machine State Register

N

NC	No Commit
NCSC	National Computer Security Center
NMI	New Machine Interface
NRL	Name Resolution List
NSA	National Security Agency

O

ODP	Open Data Path
ODT	Object Definition Table
OFCB	Output File Control Block
OIR	Object Information Repository
OMI	Original Machine Interface
OPM	Original Program Model

P

PAWA	Process Activation Work Area
PCB	Process Control Block
PCO	Process Communications Object
PCP	Prompt Control Program
PCS	Process Control Space
PD	Page Descriptor
PDT	Process Definition Template
PEP	Program Entry Procedure
PFS	Physical File System
PID	Page Identifier
PIO	Physical I/O Layer
PL	Positional List
PLMI	Programming Language Machine Interface
PLMP	Programming Language Machine Programming
POP	Prompt Override Program
POSIX	Portable Operating System Interface
PP	Page Protect
PTF	Program Temporary Fix

Q

QDT	Query Definition Template
QSR	Quarterly Status Report

R

RAMP	Ratings Maintenance Phase
RCO	Responsible Corporate Officer
REXX	Restructured Extended Executor Language
RISC	Reduced Instruction Set Computer
RM-Plan	Ratings Maintenance Plan
RMR	Ratings Maintenance Report
ROS	Read Only Storage
RPG	Report Program Generator
RR	Repeatable Read
RRCB	Request Response Control Block
RS	Read Stability
RSA	Rivest Shamir Adleman
RSM	Resident Storage Management

S

SAG	Shared Access Group
SA-Team	Security Analysis Team
SBA	SPD Bus Architecture
SBEC	Single Bit Error Correct
SCB	Spool Control Block
SCB	Stack Control BLock
SCSI	Small Computer Standard Interface
SCU	Storage Control Unit
SCV	System Call Vectored
SD	System Domain
SDIS	System Domain Inherit State
SDR1	Storage Description Register 1
SDSS	System Domain System State
SDUS	System Domain User State
SEU	Source Entry Utility
SFUG	Security Features User's Guide
SID	Segment Identifier
SLIC	System Licensed Internal Code
SM	Storage Management
SNA	Systems Network Architecture
SNA-T3	SNA Logical Unit Type 3
SPD	System Products Division
SPOOL	Simultaneous Peripheral Operations On-Line
SPT	Software Page Table
SQL	Structured Query Language
SRM	Send/Receive Message
SRQ	Send/Receive Queue
SSD	Source Sink Data
SSR	Source Sink Request
SVAL	System Value

T

TA	Technical Assessment
TCB	Trusted Computing Base
TCE	Translate Control Element
TCSEC	Trusted Computer System Evaluation Criteria
TDE	Task Dispatching Element
TDQ	Task Dispatching Queue
TFM	Trusted Facility Manual
TLB	Translation Lookaside Buffer
TOC	Table of Contents
TPOC	Technical Point-of-Contact
TRB	Technical Review Board
TWINAX	Twin-axial

U

UD	User Domain
UDIS	User Domain Inherit State
UDSS	User Domain System State
UDUS	User Domain User State
UFCB	User File Control Block
UID	User Identification Number
UIM	User Interface Manager
UOK	Unit of Work
UR	Uncommitted Read
US	User State

V

VBPOC	Vendor Business Point of Contact
VCP	Validity Checking Program
VFS	Virtual File System
VID	SLIC Internal Design
VNODE	Virtual Node
VPD	Vital Product Data
VSA	Vendor Security Analyst

W

WA	Work Area
WCB	Work Control Block
WCBT	Work Control Block Table
WCBTE	Work Control BLock Table Entry

X

XER	Fixed-Point Exception Register
XISR	External Interrupt Status Register
XPF	Extended Program Facility

Appendix F
References and Sources

The following are the references the evaluation team used in performing their design analysis and writing of this report. Some of these references may be unavailable for public release.

[1] Dijkstra, E. W. Cooperating Sequential Processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, New York, NY, 1968.

[2] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*. Linthicum, MD, December 1985. DoD 5200.28-STD, December 1985.

[3] IBM Corporation. *AS/400 Backup and Recovery - Advanced* SC41-4305-01. Rochester, MN.

[4] IBM Corporation. *AS/400 Backup and Recovery* SC41-5304-00. Rochester, MN.

[5] IBM Corporation. *AS/400 Security - Basic* SC41-5301-00. Rochester, MN.

[6] IBM Corporation. *AS/400 System API Reference* SC41-5801-00. Rochester, MN.

[7] IBM Corporation. *DCR 31704 - C2: Integrity Changes for CA Workspaces.*[1] Rochester, MN. March 18, 1993.

[8] IBM Corporation. *DCR 31487 - C2 Certification - Parameter Validation Integrity Fix.*[1] Rochester, MN. June 6, 1993.

[9] IBM Corporation. *Discretionary Access Control Documentation* V3R2 Rochester, MN.

[10] IBM Corporation. *Security - Enabling C2* SC41-3303. Rochester, MN. 1993.

[11] IBM Corporation. *AS/400 IMPI Specification*[1] Rochester, MN.

[12] IBM Corporation. *AS/400 Software Installation* SC41-5120-00. Rochester, MN.

[13] IBM Corporation. *AS/400 Machine Interface Functional Reference ZZ70-0702-07*[1] Rochester, MN. April 9, 1997.

[14] IBM Corporation. *Philosophy of Protection Revision 11* Rochester, MN. February 1996.

[15] IBM Corporation. *AS/400 CL Reference* SC41-5722-00. Rochester, MN.

[16] IBM Corporation. *AS/400 Rating Maintenance Phase – Process* Version RMP.1997.1 Rochester, MN. January 1997.

[17] IBM Corporation. *The User's Responsibility for Security Version 2* SC41-0178-00. Rochester, MN. November 1993.

[18] IBM Corporation. *SPD I/O Bus Architecture* ROM1003-00.[1] Rochester, MN.

[1] Unavailable for public release.

[19] IBM Corporation. *AS/400 Security - Reference* SC41-5302-00. Rochester, MN. 1993.

[20] IBM Corporation. *AS/400 Integrated File System Introduction* SC41-5711-00. Rochester, MN.

[21] IBM Corporation. *V4R1 AS/400 Test Plan for the Department of Defense C2 Evaluation*. Rochester, MN.

[22] IBM Corporation. *AS/400 User Domain/System State Programs DAC Policy.*

[23] IBM Corporation. *AS/400 VMC Internals Storage Management Design.*[1] Rochester, MN. November 1, 1993.

[24] IBM Corporation. *AS/400 VMC Structures and Components Storage Management Subset Draft Version, Release 4*[1]. Rochester, MN. November 1, 1993.

[25] IBM Corporation. *AS/400 Work Management*. SC41-5306-00 Rochester, MN.

[26] IBM Corporation. *AS/400 XPF Extended Control Program Facility Component Overview and Design COD Document for Component SP.*[1] Rochester, MN. May 26, 1993.

[27] IBM Corporation. *AS/400 XPF Extended Control Program Facility Component Overview and Design COD Document for Component CA.*[1] Rochester, MN. November 29, 1993.

[28] IBM Corporation. *AS/400 XPF Extended Control Program Facility Component Overview and Design COD Document for Component CL.*[1] Rochester, MN. February 3, 1993.

[29] IBM Corporation. *Amazon User Instruction Set Architecture*[1]. Rochester, MN.

[30] IBM Corporation. *Amazon Virtual Environment Architecture*[1]. Rochester MN.

[31] IBM Corporation. *Amazon Operating Environment Architecture*[1]. Rochester, MN.

[32] IBM Corporation. *AS/400 Local Device Configuration* SC41-5121. Rochester, MN.

[33] IBM Corporation, *AS/400 System Operation* SC41-4203. Rochester, MN.

Appendix G
Security Related System Values

Security system values are system values that are used to define system-wide security settings. They can, for instance, control the number of sign-on attempts allowed at a device, how often passwords need to be changed, and the length and composition of passwords. The modification of system values are auditable and only security officers are allowed to control and change the security system values. See the section 6.5.4.1 “System Values for Auditing” on page 6-22 for details about setting the audit system values. The system values are divided into 2 categories:

- Required system values, which must be set to the required value and cannot be changed from the required value in a C2 environment
- Security relevant system values, which should be reviewed and set prior to enabling the system for use

Below are the required system values settings for C2:

- QALWOBJRST: Allow Object Restore. Controls whether system state programs and/or programs that adopt their owner’s authority are allowed to be restored. *The recommended value is *NONE.*
- QALWUSRDMN: Allow User Domain Objects in Libraries and Directories. Controls which libraries are allowed to contain user domain objects of type *USRSPC, *USRIDX, and *USRQ and controls whether *SOM user domain objects are allowed in directories. *The required value is QTEMP.*
- QATNPGM: Attention Program. Specifies the name and library where the attention program resides. *The required value is *NONE.*
- QAUTOVRT: Automatic configuration for virtual devices. Controls the number of virtual devices that the user wants to have automatically configured. *The required value is 0.*
- QCRTAUT: Create Default Public Authority. Controls the public authority for newly created objects if the following conditions are met:
  - The create authority (CRTAUT) for the library of the new object is set to *SYSVAL.
  - The new object is created with public authority (AUT) of *LIBCRTAUT. See section 6.5.4.1 “System Values for Auditing” on page 6-22 for more details.

  The required value is *EXCLUDE.
- QPASTHRSVR: Pass-through Servers. The number of target display station pass-through server jobs that are available to process AS/400 display station pass-through, AS/400 Client Access workstation function (WSF), and other 5250 emulation programs on programable workstations. *The required value is 0.*
- QPWDVLDPGM: Password validation program. Controls whether a user-written program can be used to provide additional validation of the password. *The required value is *NONE.*

- QRETSVRSEC: Retain server security information. Controls whether decryptable passwords are allowed to be stored. *The required value is 0.*
- QRMTSIGN: Remote signon. Controls whether a remote sign-on is possible. *The required value is *REJECT.*
- QRMTSRVATR: Remote service attribute. Controls the remote system service problem analysis ability. *The required value is 0.*
- QRMTIPL: Remote IPL. Controls whether a remote IPL is possible. *The required value is 0.*
- QSECURITY: Security level. Controls the security level of the system. *The required value is 50.*
- QSFWERRLOG: Software error log. Controls whether software errors should be logged by the system. *The required value is *NOLOG.*
- QSPCENV: Special environment. Controls whether the AS/400 or S/36 environment is used. *The required value is *NONE.*
- QSRVDMP: Service dumps. Controls whether service dumps for unmonitored escape messages are created. *The required value is *NONE.*
- QSYSLIBL: System library list. Controls the entries in the system library list (search paths).
  - QSYS
  - QSYS2
  - QHLPSYS
  - QUSRSYS
- QUSRLIBL: User library list. Controls the default entries in the user library list (search paths).
  - QTEMP
  - QGPL

The recommended settings for the security related system values are as follows:

- QAUDCTL: Audit Control. Controls whether auditing is performed for object and user level auditing. *The recommended value is *OBJAUD *AUDLVL.*
- QAUDENDACN: Audit End Action. Controls what action the system takes if auditing is active and the system is unable to write entries to the audit journal. *The recommended value is *PWRDWNSYS.*
- QAUDFRCLVL: Audit Force Level. Controls how often new audit journal entries are forced from memory to auxiliary (DASD) storage. *The recommended value is *SYS or 1 - 100.*
- QAUDLVL: Audit Level. Controls which type of security-related events are logged to the security audit journal (e.g., authorization failures). *The recommended values are *AUTFAIL *CREATE *DELETE *SECURITY *JOBDTA *PGMADP *PGMFAIL *SERVICE.*
- QAUTOSPRPT: Automatic system disabled reporting. Controls the automatic problem reporting ability. *The recommended value is 0.*
- QCRTOBJAUD: Create object auditing. Controls the default auditing value for new objects created in a library when the auditing default for the library of the new object is set to *SYSVAL. *The recommended value is *NONE.*
- QDSCJOBITV: Disconnect job interval. Controls if and when the system ends a disconnected interactive job. *The recommended value is 5.*

- QDSPSGNINF: Display Sign-On Information. Controls whether the sign-on information display is shown after signing on. *The recommended value is 1.*
- QFRCCVNRST: Force conversion on restore. Controls whether to convert program objects during the restore. *The recommended value is 1.*
- QHSTLOGSIZ: History log size. Controls the maximum number of records for each version of the history log. *The recommended value is 32.*
- QINACTITV: Inactive job time-out. Controls in minutes how long the system allows a job to be inactive before taking action. *The recommended value is 5.*
- QINACTMSGQ: Interactive message queue. Controls the location (name and library location) of the queue and what action the system takes when the inactive job time-out interval for a job has been reached. *The recommended value is *ENDJOB.*
- QIPLTYPE: IPL type. Controls the type of IPL performed when the system is powered on manually with a key in the normal position. *The recommended value is 0.*
- QJOBMSGQFL: Job message queue full. Controls whether the job message queue should be allowed to wrap. *The recommended value is *NOWRAP.*
- QLMTDEVSSN: Limit Device Sessions. Controls whether a user is allowed to be signed on to more than one device at a time. *The recommended value is 1.*
- QLMTSECOFR: Limit security officer. Controls whether a user with *ALLOBJ or *SERVICE special authority needs explicit authorization to sign-on to any workstation. *The recommended value is 1.*
- QMAXSIGN: Maximum Sign-On Attempts. Controls the number of consecutive incorrect sign-on attempts allowed. *The recommended value is between 1 and 5.*
- QMAXSGNACN: Maximum sign-on action. Controls what action the system takes when the maximum number of sign-on attempts is reached at a workstation. *The recommended value is 3.*
- QPWDEXPITV: Days password valid. Controls the number of days before which a password expires. *The recommended value is 30.*
- QPWDLMTCHR: Limit characters. Controls the use of certain characters in a password. *The recommended value is aeiou#$@*
- QPWDLMTAJC: Limit adjacent digits. Controls the use of numeric characters next to each other in a password. *The recommended value is 1.*
- QPWDLMTREP: Limit repeat characters. Controls the use of repeating characters in a password. *The recommended value is 1.*
- QPWDPOSDIF: Limit character position. Controls each character position in a new password. *The recommended value is 1.*
- QPWDMAXLEN: Maximum password length. Controls the maximum length of a password; the maximum allowed is 10. *The recommended value is between 5 and 10.*
- QPWDMINLEN: Minimum password length. Controls the minimum length of a password; the minimum allowed is 1. *The recommended value is between 1 and 10.*
- QPWDRQDDIF: Duplicate password. Controls whether the password must be different than the last 32 passwords. *The recommended value is 5.*

- QPWDRQDDGT: Required password digits. Controls whether a numeric character is required in a new password. *The recommended value is 1.*
- QPWRRSTIPL: Power restore IPL. Controls whether the system will automatically do an IPL when power is restored after a power failure. *The recommended value is 0.*
- QRCLSPLSTG: Reclaim spool storage. Controls the automatic removal of empty spool database members. See section 4.1.3.4 “Spool” on page 4-35 for more information. *The recommended value is 12.*
- QUSEADPAUT: Use adopted authority. Controls who can create programs with the attribute “Use adopted authority - *YES.” *The recommended value is *NONE.*

Appendix H
System Jobs

AS/400 has the following types of system jobs:

- System Arbiter - The System Arbiters handle various system events. The System Arbiters are started at IPL as part of the Start CPF job. The system arbiters do not control other jobs. The arbiters are responsible for creating additional job structures, such as the WCBTE and WCB, as they run out.
- StartCPF - The Start CPF job is the first job started at IPL. It is always active while the system is running.
- Subsystem Monitor - There is one subsystem monitor job running per subsystem. When a subsystem is started, the subsystem monitor job attempts to allocate all display devices that have been configured in the subsystem description object. If there are no display devices configured in the subsystem description object, then the subsystem still starts but there will be no display devices allocated (The subsystem can still run batch jobs in this subsystem provided subsystem description object is configured to allow batch jobs). QINTER is an example of a Subsystem monitor that is provided by IBM as the default subsystem for running interactive jobs. The subsystem monitor is responsible for controlling other jobs that run within its subsystem. The Controlling Subsystem is the interactive subsystem that starts at IPL. It is the subsystem through which the system operator controls the system. There is only one controlling subsystem active on the system at any time. The default controlling subsystem handles both interactive and batch jobs.
- Database Server Jobs - These system jobs are used by the Database component. See section 4.2 “Database Management System” on page 4-68 for a discussion of functionality.
- Data Compression Jobs - This system job is used during IPL to decompress parts of the operating system. See section 4.3.1 “Initial Program Load (IPL) and Installation” on page 4-85 for a discussion of functionality.
- Performance Adjustment Jobs - This system job is started at IPL in response to a request for automatic tuning. Once started, this job wakes up every 20 seconds to look at the storage pool assignments. It then makes adjustments to the size and activity level of the storage pools. Automatic tuning will not adjust the storage pool assignments. See section 4.3.1 “Initial Program Load (IPL) and Installation” on page 4-85 for more discussion of this job.
- Spool Maintenance Job - This job is part of the Spool Component. See section 4.1.3.4 “Spool” on page 4-35 for a discussion of its functionality.
- Job Scheduler - The Job Scheduler system job is used to manage batch jobs with a particular start time.
- Alert Job - This job is used to process system alerts. Alerts are normally set up to inform operators at remote locations on the network of a particular event. The evaluated configuration supports only a stand-alone configuration. Therefore, if a log message is set up as an alert, it is not distributed.

- WCBT Cleanup Job - Cleans up WCBT at IPL.
- File System Job - Supports the background processing of the file system.
- Database Parallelism Job - Performs asynchronous database processing for the DB2 Multisystem.

Appendix I
IBM-Supplied User Profiles

The AS/400 system contains a number of IBM-supplied user profiles. The majority of these user profiles are used as object owners for various system operations. Others are used as templates to create other profiles. Also, some operations run under IBM-supplied user profiles. The following is a list of IBM-supplied user profiles:

- QAUTPRF: used to provide access to QUSRSYS - has *CHANGE authority to QUSRSYS.
- QBRMS: used by a system management LP - has no special authorities.
- QDBSHR: Database share profile - Owner of the internal database objects created during database processing. QDBSHR has no special authorities.
- QDFTOWN: Default owner profile - Used during restore when the owner of the object on the media does not exist on the system. QDFTOWN becomes the owner of these objects during the restore operation. QDFTOWN has no special authorities.
- QDOC: Document profile - Owner of the objects used by the office products. QDOC has no special authorities.
- QDSNX: Distributed systems node executive profile - Used as the process user profile of jobs that are used to route requests through the intermediate AS/400 node. QDSNX has no special authorities.
- QFNC: Finance profile - Owner of the finance product objects. QFNC has no special authorities.
- QGATE: VM/MVS bridge profile - Used as the process user profile of the gateway sender jobs of SNADS. QGATE has no special authorities.
- QLPAUTO: Licensed program automatic install profile - Used during an auto install of the system. QLPAUTO has the following special authorities: *ALLOBJ, *JOBCTL, *SAVSYS, and *SECADM.
- QLPINSTALL: Licensed program install profile - Used as the process user profile of the licensed program product install job. QLPINSTALL has the following special authorities: *ALLOBJ, *JOBCTL, *SAVSYS, and *SECADM.
- QMSF: used for mail server framework - has no special authorities.
- QNETSPLF: used to handle network spool files - has no special authorities.
- QNFSANON: used with Network File System functions - has no special authorities
- QPGMR: Programmer profile - Used as a template to create other programmer user profiles. QPGMR has the following special authorities: *JOBCTL and *SAVSYS.
- QRJE: Remote job entry profile - Used as the process user profile for remote job entries. QRJE has *JOBCTL special authority.

- QSECOFR: Security officer profile - Used when performing critical tasks on the system where (special) authority to many objects is required. It can also be used as a template to create other security officer user profiles. QSECOFR has the following special authorities: *ALLOBJ, *SECADM, *JOBCTL, *SPLCTL, *SAVSYS, *SERVICE and *AUDIT.
- QSNADS: SNA distributed services profile - Used as the process user profile of the SNADS jobs. QSNADS has no special authorities.
- QSPL: Spool profile - Owner of the database files used to hold spooled output files and inline data. This profile also owns several operating system programs used by the spool component. QSPL has no special authorities.
- QSPLJOB: Spool job profile - Used as the process user profile of the spool reader and writer. QSPLJOB has no special authorities.
- QSRV: Service profile - Used to perform service related activities that do require modifications of data via the service utilities. QSRV has the following special authorities: *JOBCTL and *SERVICE.
- QSRVBAS: Service basic profile - Used to perform service related activities that don't require modification of data via the service utilities. QSRVBAS has *JOBCTL special authority.
- QSYS: System profile - Owner of most supplied operating system objects. QSYS has the following special authorities: *ALLOBJ, *SECADM, *JOBCTL, *SPLCTL, *SAVSYS, *SERVICE and *AUDIT.
- QSYSOPR: System operator profile - Used as a template to create other system operator user profiles. QSYSOPR has the following special authorities: *JOBCTL and *SAVSYS.
- QTCP: Transmission control protocol profile. QTCP has the following special authorities: *JOBCTL.
- QTFTP: Trivial File Transfer Protocol profile - has no special authorities.
- QTSTRQS: Test request profile - Used when the test request key is used from the interactive display to run the job started by the test request function. QTSTRQS has no special authorities.
- QUSER: Workstation user profile - Used as a template to create other end users on the system. QUSER has no special authorities.