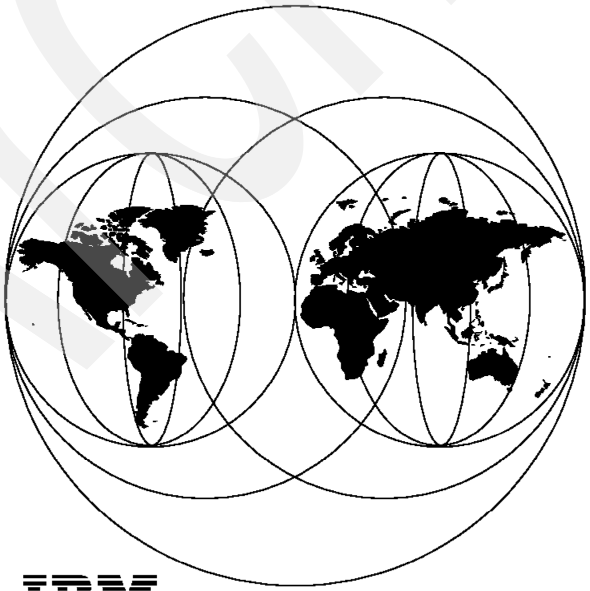
SG24-4438-00

International Technical Support Organization

UNIX C Applications Porting to AS/400

December 1995



IBM

International Technical Support Organization
Rochester Center

SG24-4438-00

International Technical Support Organization

UNIX C Applications Porting to AS/400

December 1995

Take Note!

Before using this information and the product it supports, be sure to read the general information under “Special Notices” on page xv.

First Edition (December 1995)

This edition applies to V3R6 of OS/400, Program Number 5716-SS1.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader′s feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. 977 Building 663-3
3605 Highway 52N
Rochester, Minnesota 55901-7829

Abstract

This document is unique in its detailed coverage of openness features of the AS/400 system in terms of writing UNIX C style applications. It focuses on portability issues of AS/400 programming with UNIX applications.

The AS/400 system provides many functions and features to facilitate the porting of UNIX C applications. In spite of its rich functional offerings, there are some occasions where the transparent porting is not available due to fundamental differences between UNIX and the AS/400 system. This book provides valuable information such as tips and techniques, along with examples, on how to get around if you encounter these cases in addition to the native ways of more straightforward porting cases.

This document is written to help customers, business partners, and IBM specialists in writing or porting UNIX C style applications for the AS/400 system. Some knowledge of application development, UNIX C, and ILE C/400 is assumed.

(296 pages)

Contents

Figures

Tables

Special Notices

This publication is intended to help customers, business partners, and IBM specialists in writing or porting UNIX C style applications for the AS/400 system. The information in this publication is not intended as the specification of any programming interfaces that are provided by OS/400, 5716-SS1, and Common Programming APIs. See the PUBLICATIONS section of the IBM Programming Announcement for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Preface

This document is intended to help the customers, business partners, and IBM specialists in writing or porting UNIX C style applications for the AS/400 system.

How This Document is Organized

The document is organized as follows:

- Chapter 1, "Introduction"

  This chapter describes the general purpose of this book. It describes why the AS/400 platform is open and why it can give you the benefits in writing or porting UNIX C style applications for the AS/400 system.

- Chapter 2, "Architecture of the AS/400 System"

  This provides the general concepts of the AS/400 system. If you are not yet familiar with the AS/400 system, this chapter gives you a good overview of the system that is beneficial when going on to the following chapters.

- Chapter 3, "File System - AS/400 Integrated File System"

  This chapter describes the file systems available on the AS/400 system focusing on the integrated file system.

- Chapter 4, "Process Management"

  This chapter describes the issues related with processes such as jobs, threads, spawns, and so on. It also describes how to start and stop the processes.

- Chapter 5, "Networking"

  This provides the networking features of the AS/400 system: its TCP/IP support, sockets, and so on.

- Chapter 6, "Development Environment on AS/400 System"

  This chapter describes the topics of the applications development environment on the AS/400 system from the viewpoint of UNIX C applications porting.

Appendixes of this document include:

- Appendix A, "Integrated File System Tutorial"

  This is the tutorial of the AS/400 integrated file system for the first time users.

- Appendix B, "Integrated File System Example Programs"

  This is the collection of the sample programs related to the AS/400 integrated file system

- Appendix C, "Development Cycle of ILE C/400 Applications"

  This is the tutorial of the AS/400 applications development environment and the ILE C/400 for the first time users.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *Integrated File System Introduction*, SC41-4711-00
- *System API Reference*, SC41-4801-00
- *Common Programming APIs Toolkit/400*, SC41-4802-00
- *ILE Concepts*, SC41-3606
- *ILE C/400 Programmer's Guide*, SC09-1820
- *ILE C/400 Programmer's Reference*, SC09-1821
- *ILE C/400 Reference Summary*, SX09-1288
- *AS/400 Work Management*, SC41-3306-00

International Technical Support Organization Publications

A complete list of International Technical Support Organization publications, known as redbooks, with a brief description of each, may be found in:

To get a catalog of ITSO redbooks, VNET users may type:

```
TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
```

A listing of all redbooks, sorted by category, may also be found on MKTTOOLS as ITSOPUB LISTALLX. This package is updated monthly.

How to Order ITSO Redbooks

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-445-9269. Almost all major credit cards are accepted. Outside the USA, customers should contact their local IBM office.

Customers may order hardcopy ITSO books individually or in customized sets, called GBOFs, which relate to specific functions of interest. IBM employees and customers may also order ITSO books in online format on CD-ROM collections, which contain redbooks on a variety of products.

ITSO Redbooks on the World Wide Web (WWW)

Internet users may find information about redbooks on the ITSO World Wide Web home page. To access the ITSO Web pages, point your Web browser (such as WebExplorer from the OS/2 3.0 Warp BonusPak) to the following:

```
http://www.redbooks.ibm.com/redbooks
```

IBM employees may access LIST3820s of redbooks as well. Point your web browser to the IBM Redbooks home page:

```
http://w3.itsc.pok.ibm.com/redbooks/redbooks.html
```

Acknowledgments

This project was designed and managed by:

Yessong Johng
International Technical Support Organization, Rochester Center

The authors of this document are:

Johan Helstrom
IBM Sweden

Yoon Se Yeon
IBM Korea

Park Deok Jin
IBM Korea

Charlie Quigg
IBM Rochester

This publication is the result of a residency conducted at the International Technical Support Organization, Rochester Center.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Mike Mundy
Terry O′Brien
Fred Kulack
Scott Forstie
Scott Moore
Jeff Parker
Ray Bills
Tom McBride
Kay Tate

Standard Conventions

Several conventions appear in this redbook to make it easier for you to use.

- **Boldface**

  Choices made from the actual screen or to emphasize the character strings or values on the actual screen.

- *Italics*

  For emphasizing single words.

- `Monospace`

  What a user would type on the terminal or function calls (APIs).

- UPPERCASE

  Commands, parameters, device names, file names.

This redbook also adopts the following general conventions.

MB	Mega bytes
Mb	Mega bits
Mbps	Mega bits per second
Kbps	Kilo bits per second

Chapter 1. Introduction

AS/400 System Is An Open System

You can port many UNIX applications to the AS/400 system very easily. You can write your new AS/400 applications or modernize your existing AS/400 applications and do this in UNIX style.

The AS/400 system supports many industry de facto and de jure open standards. This helps you write UNIX style applications on the AS/400 system or port many of them to the AS/400 system easily. Don′t get me wrong. I am not saying the AS/400 system is an open system because it supports many UNIX features but it supports many open standards which, coincidentally, are supported by many UNIX platforms. Right or wrong, why would you care? As program developers, IT managers, or heads of companies, you want to minimize the porting efforts of your existing applications to another platform and the AS/400 system gives you ability if you use it as your porting target system.

Let′s face it. Worldwide installations of the AS/400 system exceed 350,000. This must be the largest number of installation set among business application servers and the number is growing rapidly. If you are applications vendors, this means one successful application product on this particular platform guarantees the success of your business. If you sell your product to 10% of the AS/400 customers, it means over 35,000 licenses are sold.

1.1 AS/400 System: Open System

No Marketing Brochure

Of course, this book is not a marketing brochure but it gives you solid and very helpful tips and techniques for when you actually do port your UNIX C applications to the AS/400 system.

Nevertheless, it won′t hurt you too much just to briefly review why we say the AS/400 is an open system and how we should interpret that in the context of porting UNIX C applications.

1.1.1 What Is Open System for UNIX C Developers?

We expect you, the readers of this book, are either UNIX C application developers or have the experience. Your applications do not have to be coded in C to qualify as porting objects but this book concentrates on those applications written in C. After all, that is the most common language in the UNIX world. So, as UNIX C application developers, what is an open system anyway?

We expect that you have worked with UNIX for many years and that you have seen several UNIX manuals. Therefore, we are not going to give you the common introductory information that is in almost every UNIX manual, its history, its architecture, and so on. Well, after all this is not a UNIX manual. Rather, we list the characteristics of the UNIX system from the perspective of the application developers using C language.

- Directory Perspective
  - Hierarchical structure
  - Current and home directory
  - Search path
  - Hard and symbolic links
  - Authorizations for user, group, and other
- File and I/O Perspective
  - File descriptors
  - Unformatted streams
  - Terminal (TTY) raw mode and line mode
  - Pipes
- Process Perspective
  - The fork() and exec()
  - Signals
  - Session groups and process groups
  - Environment variables
  - Per-process address space
- Shells/Utilities Perspective
  - The sh, ls, awk, grep, and so on

  - File redirection
  - Job control
- Real Time Support Perspective
  - Threads
  - Mutexes
  - Semaphores
  - Shared memory
  - IPC message queues
- C Interface Perspective
  - ANSI C compliant
  - POSIX.1 compliant
- Networking Perspective
  - TCP/IP
  - Sockets
  - NFS

Would you agree that the AS/400 system is open enough and qualified to be a server of choice if the AS/400 system supports all of these? To rephrase the question, would you be interested in porting your current UNIX C applications to the AS/400 system if we support all of these? Or do you want to modernize your current AS/400 applications and do it in the UNIX way?

We wish we could simply say ″Yes!″ to these questions but it is not that simple. Well, if the answer is a simple yes, then you do not need the help of this book at all. But what is important is the answer could be 80% yes in general as of V3R6 of the OS/400 and it can be even much higher in a practical sense. Remember, we are positioning the AS/400 system as a *Server*-of-Choice in a *commercial* applications environment. The more your UNIX C applications are for server functions and the more your application type is for commercial use, this ratio of compatibility can be very close to 100%.

1.1.2 How Open Is AS/400 System for UNIX C Developers?

We provide the answers to this question in two ways. First, we map the AS/400 features to the topics of section 1.1.1, "What Is Open System for UNIX C Developers?" on page 2. Then, we summarize the open standards supported by the AS/400 system.

1.1.2.1 AS/400 Implementations of UNIX Features

The AS/400 implementations of UNIX features follow the three major open standards:

- *Single UNIX Specification*, formerly *Spec1170*
- POSIX.1
- OFS/1 for UNIX style file system implementation

Table 1. AS/400 System's Directories Features Summary

UNIX Feature	**AS/400 Implementation**
Hierarchical structure	Yes - in IFS*
Current and home directory	Yes - in IFS*
Search path	Yes - in IFS*
Hard and symbolic links	Yes - in IFS*
Authorizations for user, group, and other	Yes - in IFS*

Table 2 AS/400 System's File and I/O Features Summary

Table 2. AS/400 System's File and I/O Features Summary

UNIX Feature	**AS/400 Implementation**
File descriptors	Yes - in IFS*
Unformatted streams	Yes - in IFS*
Terminal (TTY) raw mode and line mode	No
Pipes	Yes - in OS/400

Table 3 on page 5 AS/400 System's Process Features Summary

Table 3. AS/400 System's Process Features Summary

UNIX Feature	AS/400 Implementation
fork() and exec()	No direct support but the alternatives are spawn(), SBMJOB, and threads
Signals	Yes (synchronous signals are in ILE C/400 and asynchronous signals are in CPA ToolKit)
Session groups and process groups	Yes
Environment variables	Yes
Per-process address space	Yes

Table 4 AS/400 System's Shells/Utilities Features Summary

Table 4. AS/400 System's Shells/Utilities Features Summary

UNIX Feature	AS/400 Implementation
sh, ls, awk, grep, and so on	No (alternatives can be CL, CLP, and REXX)
File redirection	No (alternatives can be CL, CLP, and OVRDBF)
Job control	Yes (partial support)

Table 5 AS/400 System's Real Time Support Features Summary

Table 5. AS/400 System's Real Time Support Features Summary

UNIX Feature	AS/400 Implementation
Threads	Yes (in CPA ToolKit)
Mutexes	Yes (in OS/400)
Semaphores	Yes (in CPA ToolKit)
Shared memory	Yes (in CPA ToolKit)
IPC message queues	Yes (in OS/400)

Table 6 on page 6 AS/400 System's C Interface Features Summary

Table 6. AS/400 System′s C Interface Features Summary

UNIX Feature	AS/400 Implementation
ANSI C compliant	100% (in ILE C/400)
POSIX.1 compliant	75% (in IFS)

Table 7 AS/400 System′s Networking Features Summary

Table 7. AS/400 System′s Networking Features Summary

UNIX Feature	AS/400 Implementation
TCP/IP	Yes (in OS/400)
sockets	Yes (in OS/400)
NFS	Yes and No (in FSS/400)

I expect that you have noticed that some of the features are not supported but most of them are. I expect that you have noticed that those not supported yet are more for the client′s code rather than the server′s side. I also expect that you are encouraged to port your UNIX applications to the AS/400 system with those important features supported. Finally, I expect that you have strong interests to understand more details of IFS, CPA ToolKit, and ILE C/400. Good. This book is here for you.

1.1.2.2 Open Standards Supported by the AS/400 System

The AS/400 heritage is based on integration. That is, emphasis has been on providing an integrated solution including database, security, integrity, and systems management. In the last few years, we have broadened our focus to ensure that the AS/400 system is also an open system. This means that the AS/400 system conforms to industry standards, facilitates application portability, and interoperates with hardware from other vendors. Table 8 shows AS/400 support for standards in all of the categories.

Table 8 (Page 1 of 3). Standards Supported by the AS/400 System

Category	Standards
Data Access	ANSI/ISO SQL2 1992 DAL (Apple) DRDA L2 ODBC L2 (Microsoft) OSI-FS FTAM - ISO 8571 SQL FIPS 127-2 (subset)

Table 8 (Page 2 of 3). Standards Supported by the AS/400 System

Category	**Standards**
Data Interchange	EDI - ANSI X.12, ISO EDIFACT
Directory	DCE Directory (Call) SNA APPN
Distribution Services	DCA (IBM) DIA OSI-MS X.400 - ISO 10021 (1984) SMTP SNADA/ODF
File Serving	LAN Server NetWare NFS Server
Language Portability	C - ANSI X3.159, X3J11/90-013 C + + COBOL - ANSI/ISO X3.23 FIPS 151-2 (subset) Pascal - ANSI 770X3.97 PL/1 ISO POSIX 1003.1 ISO/IEC 9945-1 (subset) POSIX 1003.2 (SOD) POSIX 1003.4a (subset) POSIX 1003.4b draft 7 for process control (spawn) REXX (IBM) RPG (IBM) Single UNIX Specification (subset) Smalltalk (client) SOM/DSOM - CORBA SVR4 IPC (subset) XPG4 Base (subset)
Mail Serving	MAPI MIME VIM

Table 8 (Page 3 of 3). Standards Supported by the AS/400 System

Category	Standards
Networks	NetWare IPX/SPX OSI-CS ACSE - X.227/ISO 8650 OSI-CS Transport - ISO 8073 OSI-CS Presentation Kernel - X.226/ISO 8823 OSI-CS Presentation ASN.1 - X.208/209, ISO 8824/8825 OSI-CS Session - V1/2, X.225/ISO 8327 OSI-CS Network CLNS - ISO 8473 OSI-CS Network CONS - X.223/ISO 8878 SNA APPC (SNA LU6.2) TCP/IP FTP TCP/IP LPR/LPD TCP/IP TCP, UDP, IP, ICMP, ARP TCP/IP TELNET 3270 Passthru (IBM) 5250 (IBM)
Program-to-Program	BSD 4.3 Sockets CPI-C DCE RPC MQI
Security	C2 Certification DCE Kerberos DES Cryptography RSA Cryptography
Subnetworks	Apple Localtalk Asynchronous ATM (SOD) Ethernet - V2, ANSI/ISO 8802.3, IEEE 802.3 FDDI/SDDI - ISO 9314.2 ANSI X3T9.5 Frame Relay - ANSI T1.618, ITU 2.922 ISDN - ITU Q.931/Q.922/Q.921, National ISDN 1/ISDN 2 Euro ISDN (ETSI) NetBIOS (FSIOP) SDLC Token Ring - ANSI/ISO 8802.5, IEEE 802.5 Wireless LAN - IEEE 802.11 Wireless WAN X.25 - ITU X.25/X.31/X.32, ISO 8208, ISO 7776
System Management	NetView (IBM) SNMP Agent
Time	DCE Time
Transaction Management	CICS Tuxedo (announced)

The AS/400 strategy for implementing UNIX function has been to provide APIs and system interfaces that are most valuable to most vendors. This portability support covers APIs in these categories:

- File system
- Process control
- Interprocess communication (IPC)
- Threads
- Sockets
- TCP/IP

1.1.3 Why This Can Be So Easy on the AS/400 System?

Maybe we can better rephrase the section title as:

1. How come the AS/400 system adopts so many features that sound totally different from its original architecture?
2. If this kind of diversity promises even more success for the AS/400 system, how come the competitors don′t do the same?

The answer to both questions is the same: it is because of a unique architecture on the AS/400 system. What′s so unique about the architecture?

On the AS/400 system, the major components can be changed without redesigning the whole system. The heart of the AS/400 system′s ability to change without disrupting the customer and their applications is the Technology-Independent Machine Interface (see Figure 1 on page 10).

The AS/400 Advanced Application Architecture is unique in the industry. The AS/400 system is the only system that insulates the hardware from the software running on it. The insulating layer is the Technology-Independent Machine Interface.

This means that when changes are made to AS/400 hardware, the operating system and your application are not affected; you do not have to rewrite or even recompile your code to migrate to the new hardware. This lets you make technology advances while protecting your customer′s investments in competitive applications.

This is the secret behind the ease with which the AS/400 system keeps in step with massive leaps in storage, memory, and processor technology. This machine interface is behind the ease of the transition to a 64-bit, Power PC-based central processor. The transition to a 64-bit processor was not a complete rewrite of the operating system—far from it; the transition was with the ease of a normal release.

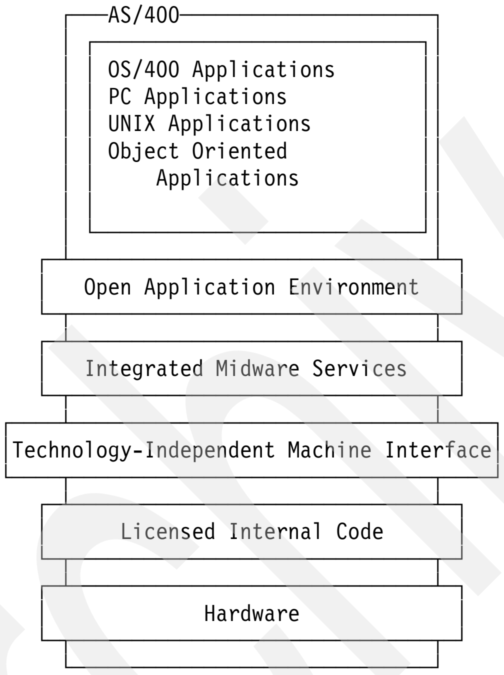

AS/400
OS/400 Applications
PC Applications
UNIX Applications
Object Oriented Applications
Open Application Environment
Integrated Midware Services
Technology-Independent Machine Interface
Licensed Internal Code
Hardware

Figure 1. AS/400 Advanced Application Architecture

Figure 2 on page 11 shows how the Technology Independent Machine Interface sets the AS/400 system apart from its competitors and how your applications are insulated from the usual effects of evolving change.

And, most importantly, this philosophy is behind the move to a leadership position in client/server computing. Your benefit is the savings in research and development, education, and support versus having to learn and relearn with every technological advancement.

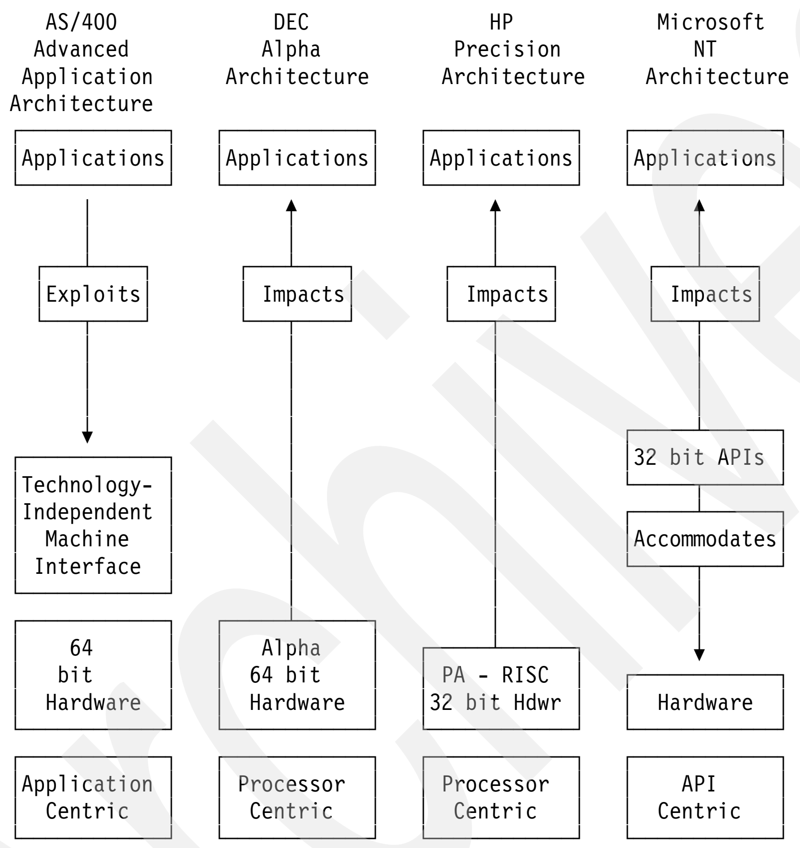


Figure 2. AS/400 Advanced Application Architecture Supports Nondisruptive Change

1.2 Evaluation of Porting Cost

Evaluating and estimating an application for porting to the AS/400 system is a complex exercise. To know what causes the most effort in doing the port, it takes both:

- Original UNIX C applications developers - who know the applications.
- An AS/400 specialist - who knows the AS/400 system.

Some ports are easy. Some applications cannot be ported. These applications require a redesign and rewrite. Many factors are involved in evaluating the port. These include:

- What is the user interface? A graphical user interface (GUI) naturally poses more problems than a simple green-screen interface. A GUI possibly poses the biggest porting challenge.
- Does application database usage include stored procedures or triggers? In some databases (Oracle, Sybase, Informix), these are written in proprietary SQL languages. They must be rewritten for DB/2 for OS/400.
- Are the application′s SQL calls ANSI/SQL compliant? DB/2 for OS/400 is fully compliant with the ANSI standard. Non-standard calls must be rewritten.
- Is the application′s current platform dependence high or low? A basic truth applies here as with any port. Dependency on a specific platform increases the complexity of the port.
- Is this a client/server application? Which client/server model should be used? What midware should be used?
- Does the application require database? Is it compatible with the integrated DB2 for OS/400 relational database?
- What connectivity does the application demand?
- What standards does the application comply with?
- How do the AS/400 facilities map to the application needs? The AS/400 operating system has the most-used UNIX system application program interfaces (APIs). An application that contains seldom used, obscure, or obsolete APIs has greater differences. Do you use the APIs we do not yet support?

1.2.1 Major UNIX-AS/400 Differences

There are some operating system areas that are different enough that you may have to recode application sections that rely on such areas. This section briefly describes areas that might affect your porting effort.

- Character encoding
  - UNIX - ASCII
  - AS/400 - EBCDIC. Applications that use character encoding rather than the actual character give unexpected results. For example, ′x4E′ is an N in ASCII but a + (plus sign) in EBCDIC.

- Pointer usage
  - UNIX - Address pointers are four bytes long. An address can be assigned to an integer and an integer to a pointer. Such assignments are simple copy operations and require no conversion.
  - AS/400 - Address pointers are 16 bytes long. AS/400 pointers are controlled by the hardware. Pointer arithmetic, assignment operations, casting, compares, and so forth work as expected as long as the system knows that variables are address pointers. Programming habits such as manipulating pointers as integers does not work.
- Standard stream
  - UNIX - stdin, stdout, and stderr. Some UNIX applications code a dependency on 0, 1, and 2 as respectively as file descriptors for the standard stream files.
  - AS/400 - The AS/400 stdin, stdout, and stderr are file pointers, not file descriptors. The standard stream pointers do not reserve the file identifiers, nor are they implicitly open at process initiation.
- Shell
  - UNIX - Shell programs and utilities.
  - AS/400 - Does not provide a shell. Shell programs can be written in the AS/400 Control Language (CL) or any of the languages that provide efficient functional equivalence.
- Display form handling
  - UNIX - Curses package.
  - AS/400 - Dynamic Screen Manager (DSM) APIs provide the means of handling application displays. Translating from curses to DSM is more than a simple port.
- Buffered versus unbuffered I/O
  - UNIX - Typically use unbuffered (character-by-character) I/O.
  - AS/400 - AS/400 buffers the I/O to external devices. I/O is handled by I/O processors that deal with blocks of data. Only certain I/O signals (for example, enter, escape, system request key strokes) send an interrupt to the AS/400 CPU.
- Default environment variables
  - UNIX - Available through the shell.

- AS/400 - Not directly available. One technique is to write an application to set the required environment variables and then set the initial logon program to invoke this application.

- Process control
  - UNIX - fork() and exec().
  - AS/400 - AS/400 system provides spawn() and wait() APIs that can handle most fork and exec processing.
- Asynchronous signals
  - UNIX - Supported.
  - AS/400 - A prototype is available in V3R1. Asynchronous signals are integrated into V3R6. However, the AS/400 implementation has some differences from the signals model on some UNIX systems.
- Interactive jobs
  - UNIX - When programs spawned from the shell end, all open descriptors are implicitly closed.
  - AS/400 - The application must take care to explicitly close descriptors.

1.2.2 What IS Easy and What IS Difficult?

The following applications port easily to the AS/400 system:

- Client/server types of applications that use the AS/400 system for either application or data server. User interface (terminal I/O) is handled on the client side.
- Applications that do not have a user interface
- Applications that utilize AS/400 supplied UNIX-type APIs for process control (including signals, and parent-child spawn relationships), interprocess communications, sockets, and threads all port reasonably well.

Applications that do not port well are host-centric UNIX applications that rely on raw mode terminal I/O model with ASCII attached devices. In many cases, these require a rewrite.

1.3 What Is This Book All About?

It is very important to understand the intention of this book correctly. I think there are two important things you have to understand in porting your UNIX C applications to the AS/400 system:

- What is supported on the AS/400 system in a rather native way.
- What is not supported.

For what is not supported, you might want to have a certain guideline in place to determine if you want to whether to find a way to work around what is not supported or re-write the code.

This book is more for the second point. For what is supported, you can find the information in the regular AS/400 manuals. We do our best to direct you to the relevant source of the information for what is supported.

We hope you are not discouraged if what is covered in this book sounds complicated. That is the purpose, in a way, of this book. Most of the part of your porting work is handled by the AS/400 system; almost automatically.

Chapter 2. Architecture of the AS/400 System

This chapter discusses certain architectural characteristics of the AS/400 system. Aspects considered are those of interest to a UNIX C application developer. Further, this chapter discusses characteristics of a UNIX operating system as related to the computer architecture.

2.1 Architecture of AS/400 System

The term computer architecture has a somewhat different meaning for the AS/400 system compared to how the term is traditionally used in computer science. AS/400 system architecture is defined by a fairly high-level machine interface (MI).

Note: In Version 3 Release 6, the MI is sometimes referred to as a Technology Independent Machine Interface (TIMI). The ″Technology Independence″ refers to the change from a CISC-based processor to a RISC-based processor. The MI layer is a boundary (actually a set of instructions) that separates the hardware and Licensed Internal Code from the operating system (OS/400). This permits the machine instructions to be rather generic and machine (hardware) independent. Dependencies have been absorbed by internal microcode (Licensed Internal Code, or LIC).

This section discusses the following architectural aspects of the AS/400 system:

- Object oriented architecture
- Addressing/storage management
- Contexts (libraries) and address resolution
- User profiles and authority management
- Character sets and terminal I/O

The particular emphasis is on distinctions from what one might expect of an architecture supporting a UNIX operating system.

2.1.1 Object Oriented Architecture

Objects are the means through which information is stored and retrieved on the AS/400 system. This concept is different from the typical byte-string manipulation of many systems. Object orientation is part of the architecture and affects both operating system implementation and high level language interaction with the system. The motivation for the Open Blueprint design is improved system integrity, reliability, and authorization as is discussed. Further, the implication for high level language users is discussed.

As previously mentioned, the MI is a boundary (set of instructions) that separates the hardware and Licensed Internal Code (LIC) from the operating system. For Version 3 Release 1 and earlier releases, the hardware and LIC implement a lower level instruction set called the Internal MicroProgrammed Interface (IMPI). LIC was implemented using IMPI instructions and contained the traditional kernel-type functions such as storage management, resource management, authority checking, and so on. For V3R6, the Licensed Internal Code was rewritten using C++. Because of the MI boundary, the LIC rewrite had little effect on the operating system and high level language.

Objects have operational characteristics and have a defined set of operations that can be performed on them. Objects are addressed through 16-byte pointers (8 bytes are used for an MI address; the other 8 bytes are used for information about the object pointed to, and for reserved space). In addition to providing addressability to the object, pointers provide access to the associated storage, data integrity, and security. Above the MI, the contents of the pointer are encapsulated.

Below the MI, the licensed internal code provides a tag bit for each quadword (16 bytes which must be aligned on a 16-byte boundary) within main storage. This bit is not addressable by the normal LIC instructions used to address storage. The bit identifies quadwords in storage containing MI pointers. Programs above the MI have no direct access to the tag bit. The tag bit is turned on by the LIC when a pointer is set and turned off by the hardware anytime the quadword is modified (except through a controlled set of LIC pointer manipulation instructions). This procedure allows the system to detect invalid pointers and prevent illegal use of a pointer. An attempt to subsequently use this data as a pointer results in an exception; the instruction is not completed. It is not possible to counterfeit a pointer or to modify a pointer in an invalid way.

The tag bit implementation allows the validation of pointers in an extremely efficient way and is the basis for system and data integrity since pointers can contain authorization information as well as addresses.

Implications for High Level Language: Pointer arithmetic, assignment operations, casting, compares, and so on all work as expected as long as the system is aware that variables are address pointers. A pointer in the ILE C/400 is 16 bytes long. Programming habits such as manipulating pointers as integers do not work.

On a UNIX system, integers and pointers are both typically 4 bytes long. An integer can be assigned to a pointer address. An address of an object of one data type can be assigned to a pointer of another data type without proper pointer casting. On the AS/400 system, such assignment operations produce exceptions. Another important implication for high level language as previously mentioned, is that pointers must be aligned on a 16-byte boundary.

2.1.2 Addressing/Storage Management

AS/400 storage management uses the idea of single level storage. With single level storage there is a single, large, uniformly addressable address space for all memory (both main storage and secondary storage). Storage is addressed by a 48-bit (6-byte) address. This large virtual address equates to 281 000GB of addressable storage.

Note: With the move to a 64-bit processor for the RISC models, the virtual address was increased to 64 bits.

There is a single page table (sometimes referred to as a page directory) that maps all virtual addresses to corresponding physical addresses. Addresses are unique across the system, not duplicated across processes. The same address in a different process points to the same storage location. This concept is different from UNIX (where there is one address space per process), and has implications for how storage is managed, and how processes are created and managed.

The UNIX System V kernel divides the virtual address space of a process into logical regions. A region is a contiguous area of the virtual address space that can be treated as a distinct space to be shared (with other processes) or protected. UNIX address spaces are per process. The same address in different processes can point to different spaces. Since the AS/400 address space is per system, the same address in different processes always points to the same space. Consequently, the way addresses are translated and the way memory is managed is fundamentally different between AS/400 architecture from that which is typically associated with UNIX systems.

These differences are summarized in Table 9.

Table 9. AS/400 and UNIX Storage Management Differences

AS/400 System	**UNIX**
Single level storage	Process address storage
Persistent addresses	Relative addresses
Single process (job)	Multiple processes
Full job structure	Lightweight process

Job/Process Structure: An example of a UNIX system call that cannot be implemented on the AS/400 system is `fork()`, which is how a process is created on a UNIX system. The UNIX kernel does (among other things) the following operations for `fork()`:

- Allocates a slot in the process table for the new process.
- Assigns a unique ID number for the child process.
- Makes a logical copy of the parent process.

The notion of copying storage (which contains pointers) of the parent process is inconsistent with the AS/400 architecture. On UNIX systems, pointers are relative to the process. On the AS/400 system, pointers are absolute because of the single address space for entire system. Consequently, `fork()`--in UNIX terms--cannot be implemented on the AS/400 system.

Don′t worry about a false alarm, though. We do not support `fork()` and `exec()` but a combination of `fork()` and `exec()` semantics has been implemented on the AS/400 system with `spawn()` and related APIs. We discuss `spawn()` in more details later. Further, the AS/400 dynamic call allows an ease-of-use call for multiple ″main″ programs while remaining in the same process. This provides single-thread dynamic execution not available in UNIX.

2.1.3 Library and Address Resolution

At the time an AS/400 object is created, the operating system places the object name in (another) machine object called a context. Contexts are presented to the user as *libraries* (not to be confused with UNIX libraries). The context object maps (resolves) the symbolic identification (type and name) of an object to its virtual addresses. A user-specified (and modifiable) list of libraries is associated with each job on the system, and objects can be referenced by the user explicitly qualified to a specific library. If not explicitly qualified to a library, the library list of the job resolves the references by searching each library in the list in order until a matching entry is found.

On UNIX systems, everything is treated as a file and addressed through hierarchical directories. The AS/400 system has objects addressed through contexts (AS/400 libraries). This notion contrasts to the concept of symbolic name resolution as used with directories and file systems on UNIX systems. UNIX only searches paths for executable files, not arbitrary files.

With V3R1 and beyond, the AS/400 system does have a similarity with UNIX by supporting a hierarchical, case-sensitive, POSIX-compliant name space (called QOpenSys). Byte stream files stored in QOpenSys are addressed through directories. The integrated file system (IFS), including QOpenSys, is discussed more later.

2.1.4 User Profile and Authority Management

System authorization management is based on user profiles that are also objects. All objects created on the system are owned by a specific user. Each operation or access to an object must be verified by the system to ensure the user′s authority. The owner or appropriately authorized user profiles may delegate to other user profiles various types of authorities to operate on an object. Authority checking is provided uniformly to all types of objects.

The object authorization mechanism provides various levels of control. A user′s authority may be limited to exactly what is needed. Files stored in QOpenSys are authorized in the same manner as UNIX files. Figure 3 on page 22 shows the relationship between UNIX permissions and security used on AS/400 database files.

Option	Control
*EXCLUDE	No access to object
*OBJOPR	Use object (outlined below)

	*OBJOPR	Data Authority				
		*READ	*ADD	*UPD	*DLT	*XEQ
r (read)	X	X				
w (write)	X		X	X	X	
X (execute)	X					X

Figure 3. Mapping UNIX Permissions to AS/400 Security

2.1.5 Character Sets and Terminal I/O

This section describes the differences between two architectures such as character sets and terminal I/O.

2.1.5.1 EBCDIC versus ASCII

Most UNIX systems run on hardware that uses ASCII character encoding. The AS/400 system uses EBCDIC encoding. This architectural difference is not a problem, assuming high level language applications do not have a dependency on (or make an assumption about) the character set. Problems arise if, for example, applications are coded with dependencies on a hexadecimal representation of character. The hex representation varies between ASCII and EBCDIC. Similarly, the collating sequence (that is, ordering of characters) also differs.

2.1.5.2 Buffered versus Unbuffered I/O

Input and output from and to external devices is buffered on the AS/400 system. I/O is handled by I/O processors that deal with blocks of data. Conversely, UNIX systems typically operate with character-by-character (unbuffered) I/O. On the AS/400 system, only certain I/O signals (for example, enter, function keys, and system request) send an interrupt to the CPU.

2.1.5.3 I/O Controllers versus Device Drivers

It may not be common, but it is possible on most UNIX systems (as well as most personal computers) to write applications to device drivers. That is, it is possible to write applications utilizing a serial port that controls an I/O device. On the AS/400 system, I/O is handled by system I/O managers (IOMs) that communicate with device controllers to handle I/O requests. Since IOMs are part of the AS/400 architecture, it is not possible to write applications directly to a particular device.

2.2 Architectural Summary

This chapter has discussed AS/400 architectural distinctions from UNIX. These differences should not affect or impede source code portability of applications to the AS/400 system (assuming that the code is written in an otherwise portable manner). Significant architectural features that uniquely identify the AS/400 system include high-level machine interface (which really defines the architecture), object-orientation, and single-level storage. The high-level machine interface permits the underlying implementation of the hardware to change without affecting users above the MI (including the operating system and end user applications). Further, these architectural features inherently provide a high degree of data and system integrity, authorization, and reliability. These features--if they exist on a UNIX system--typically must be provided by higher-level functions of the operating system or by the application.

Chapter 3. File System - AS/400 Integrated File System

This chapter explains how the UNIX style file system is implemented on the AS/400 system You will know how we do this when you are done with this chapter but if you want the answer now, it is the AS/400 system′s integrated file system. Integrated file system is new concept on the AS/400 system for V3R1 of OS/400. We want to position the AS/400 system as a *server of choice* serving the major clients: PC clients, UNIX clients, and the AS/400 clients themselves. For this, one of the first things to be done is implementing each client′s file system on the AS/400 system as native as possible.

You have two choices to store your data on the AS/400 system: in its native (and traditional pre V3R1 of OS/400), DB, or in a UNIX style file system (/ ″root″ file system or /QOpenSys file system which are two of seven file systems of integrated file system).

3.1 AS/400 Integrated File System Introduction

The AS/400 integrated file system was introduced for V3R1 of OS/400 and enhanced for V3R6. The integrated file system is a part of OS/400 that supports stream input/output and storage management similar to the UNIX operating system, while providing an integrating structure over all of the information stored on the AS/400 system.

As we mentioned earlier, the AS/400 system supports the major clients in their close-to native mode as a server of choice. It also gives additional benefits on top of having ″my own″ native file system: comprehensive and integrated management facilities over all of the data under its realm; thus, a name such as *integrated* file system. For the particular purpose of this book, we do not cover other sides of the integrated file system. We focus on the UNIX file system related features of the integrated file system. For the general information of the integrated file system, refer to *AS/400 Integrated File System Introduction*, SC41-4711-00.

Integrated file system features include:

- Hierarchical Directory Structure
  - Directory support
  - Current directory and home directory

  – Absolute and relative path name
  – Hard link and symbolic link
- Stream files
- POSIX.1 APIs that perform operations on integrated file system directories and stream files
- UNIX style permissions support for security
- Local sockets support
- Extended attributes support
- Data conversion support
- Save and restore support

As we mentioned earlier, there are seven file systems in the integrated file system. Table 10 describes each of seven file systems of the integrated file system.

Table 10 (Page 1 of 2). File Systems in the Integrated File System

File System	Description
Root	The root (/) file system. This file system is designed to take full advantage of the stream file support and hierarchical directory structure of the integrated file system. It has the characteristics of the DOS and OS/2 file system. In this file system, the file names are not case sensitive.
QOpenSys	The open systems file system. This file system is designed to be compatible with UNIX-based open system standards, such as POSIX and XPG. Similar to the / ″root″ file system, it takes advantage of the stream file and directory support provided by the integrated file system. In addition, it supports case-sensitive object names.
QSYS.LIB	The library file system. This file system supports the AS/400 library structure and provides access to database files and all of the other AS/400 system object types that are managed by the library support.

Table 10 (Page 2 of 2). File Systems in the Integrated File System

File System	Description
QDLS	The document library services file system. This file system supports the folders structure and provides access to documents and folders.
QLANSrv	The LAN Server/400 file system. This file system provides access to the same directories and files that are accessed through the LAN Server/400 licensed program. It allows users of the OS/400 file server and AS/400 applications to use the same data as LAN Server/400 clients.
QOPT	The optical file system. This file system provides access to stream data stored on optical media.
QFileSvr.400	The OS/400 file server file system. This file system provides access to other file systems that reside on remote AS/400 systems. The integrated file system of another AS/400 system can be accessed as /QFileSvr.400/RemoteSystem/...

3.2 How to Work with Integrated File System

This section explains the interface through the integrated file system. If you are already familiar with the user interface of the integrated file system, you may skip this section and go to section 3.3, "Integrated File System and Porting" on page 37. A tutorial has been provided for the first time users of the AS/400 integrated file system as the appendix Appendix A, "Integrated File System Tutorial" on page 171.

3.2.1 UNIX Commands Equivalents on AS/400 System

The AS/400 system has three types of user interfaces: commands (CL for Command Language is the name we use), menus, and displays. You are probably most familiar with the commands interface. First we cover UNIX command equivalents on the AS/400 system. Then we take a tour of the integrated file system.

3.2.1.1 Bonus Section for Real Beginners on the AS/400 System

This section is only for real beginners on the AS/400 system. If you are already working on the system, skip to section 3.2.1.2, "UNIX Commands Equivalents for Integrated File System Commands" on page 34.

Commands on the AS/400 system have naming conventions unlike UNIX commands. We take a word and abbreviate it to three characters in most cases. For example, DSPLIB is used to ″Display (DSP) Library (LIB)″, and WRKCFGSTS is used to ″Work (WRK) with Configuration (CFG) Status (STS)″. Sometimes we just take one character from a word instead of three. For example, DLTF is used to ″Delete (DLT) File (F)″.

Thanks to this generous usage of the number of characters in a single command, the AS/400 commands are very easy to guess what they do. Take the following exercise and see what I mean. What do you think the following AS/400 commands do?

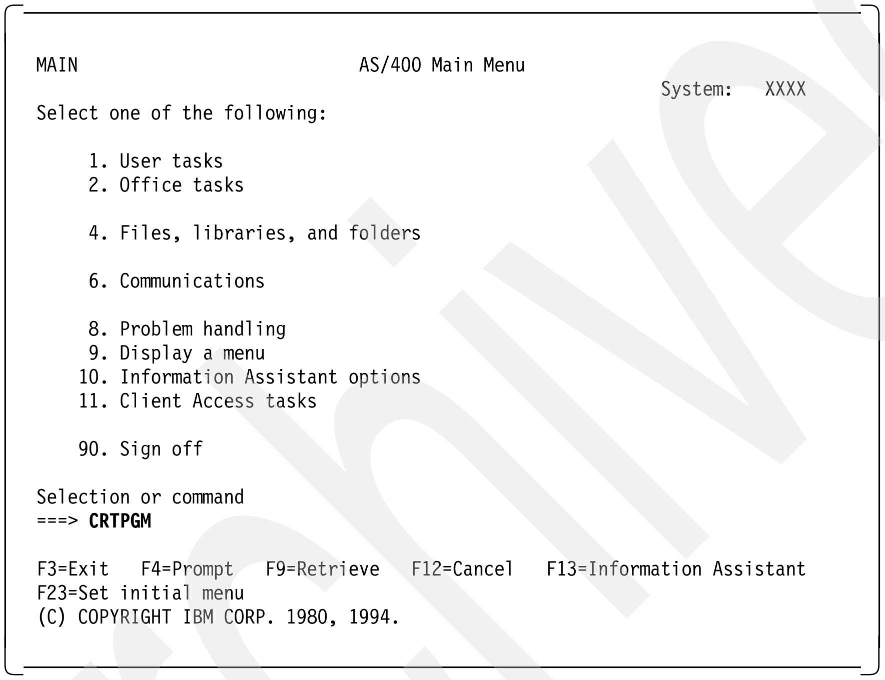
```
CRTPGM
WRKSYSSTS
WRKLNK
DSPCURDIR
CHGOWN
```

I don′t think you even need the answers but they mean Create Program, Work with System Status, Work with Link, Display Current Directory, and Change Owner.

Likewise, if you do not know the command for the function you are looking for, guessing it on the AS/400 system is equally easy. What command would you execute to delete a program? Yes, you guessed it right. It is DLTPGM.

On top of this, the AS/400 system provides the display for prompting the parameters you need for each command.

For example, let's go to the Prompt Text display of CRTPGM command. Enter CRTPGM on the command line and press <F4> key on any display.

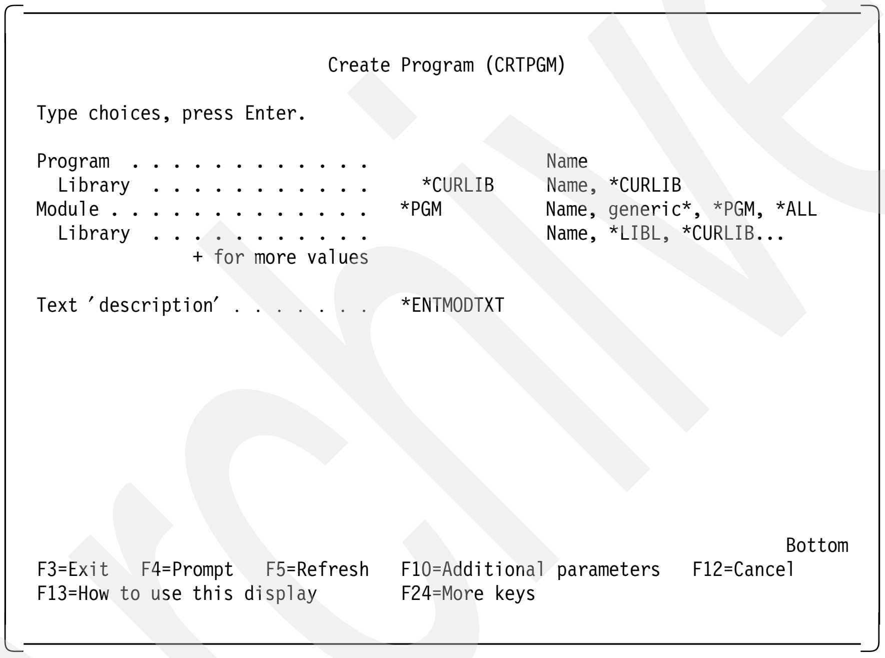


```
MAIN                           AS/400 Main Menu
                                                                System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> CRTPGM

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
(C) COPYRIGHT IBM CORP. 1980, 1994.
```

Figure 4. AS/400 Main Menu with CRTPGM Command Specified

Then the Create Program Prompt Text window is displayed. Let's check this display. The highlighted field is a mandatory field and others are optional fields. And this display provides cursor sensitive online help and question mark (?) service for value search.

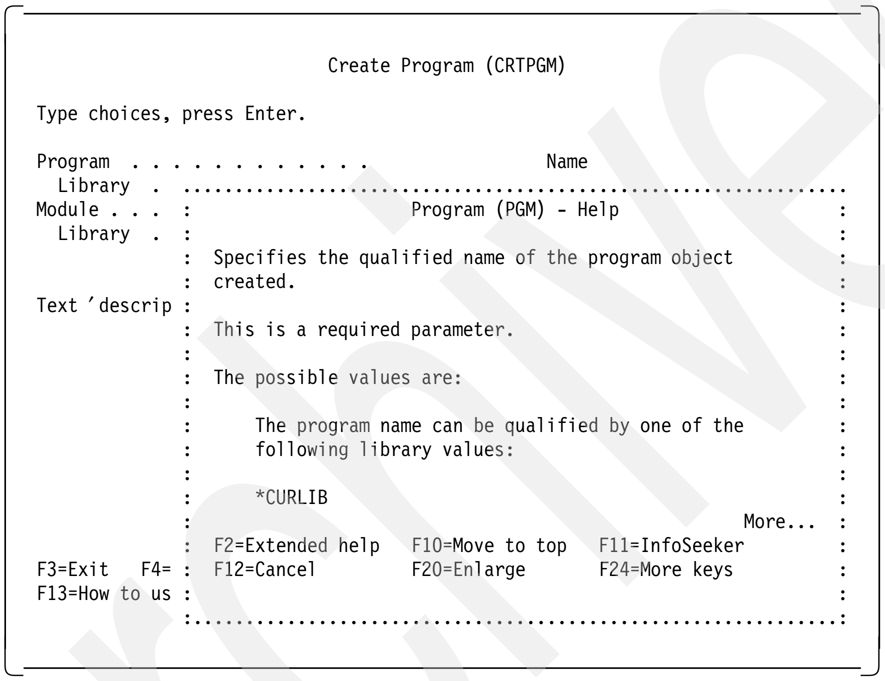
Now, the cursor must be on the first field. Press the <F1> key.



```
                            Create Program (CRTPGM)

 Type choices, press Enter.

 Program  . . . . . . . . . . . .                Name
   Library  . . . . . . . . . . .     *CURLIB     Name, *CURLIB
 Module . . . . . . . . . . . . .   *PGM          Name, generic*, *PGM, *ALL
   Library  . . . . . . . . . . .                 Name, *LIBL, *CURLIB...
               + for more values

 Text 'description' . . . . . . .   *ENTMODTXT











                                                                         Bottom
 F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel
 F13=How to use this display        F24=More keys
```

Figure 5. Create Program Prompt Text

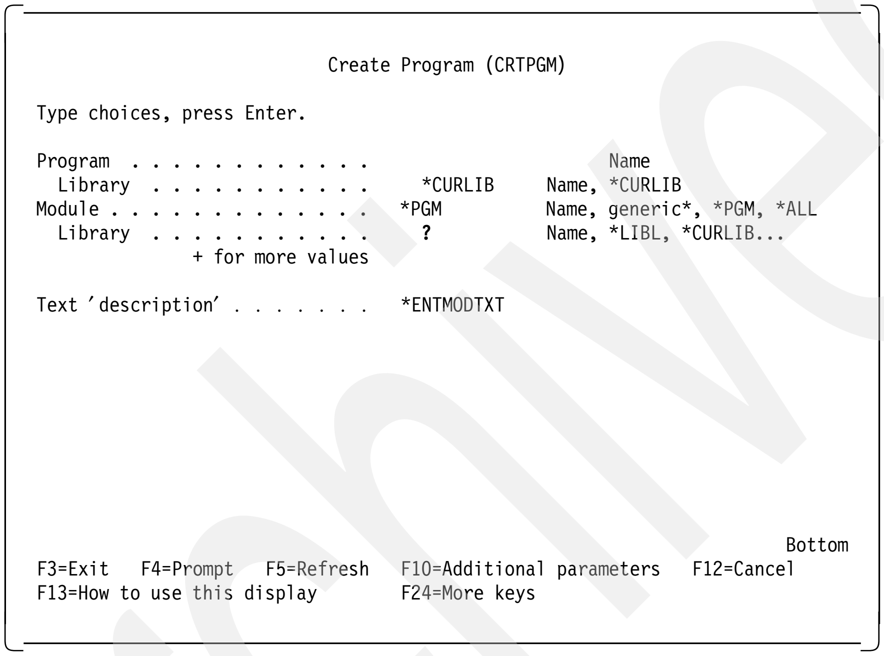
You can see the help window for the field on which the cursor is positioned. When the cursor is not positioned on the field, then you can get the help for this entire window.



```
                          Create Program (CRTPGM)

Type choices, press Enter.

Program  . . . . . . . . . . .                  Name
  Library  .  ................................................................
Module . . .  :                    Program (PGM) - Help                     :
  Library  .  :                                                             :
              :  Specifies the qualified name of the program object         :
              :  created.                                                   :
Text ′descrip :                                                             :
              :  This is a required parameter.                              :
              :                                                             :
              :  The possible values are:                                   :
              :                                                             :
              :      The program name can be qualified by one of the        :
              :      following library values:                              :
              :                                                             :
              :      *CURLIB                                                :
              :                                                     More... :
              :  F2=Extended help   F10=Move to top   F11=InfoSeeker        :
F3=Exit   F4= :  F12=Cancel         F20=Enlarge       F24=More keys         :
F13=How to us :                                                             :
              :.............................................................:
```

Figure 6. Help of Create Program Prompt Text

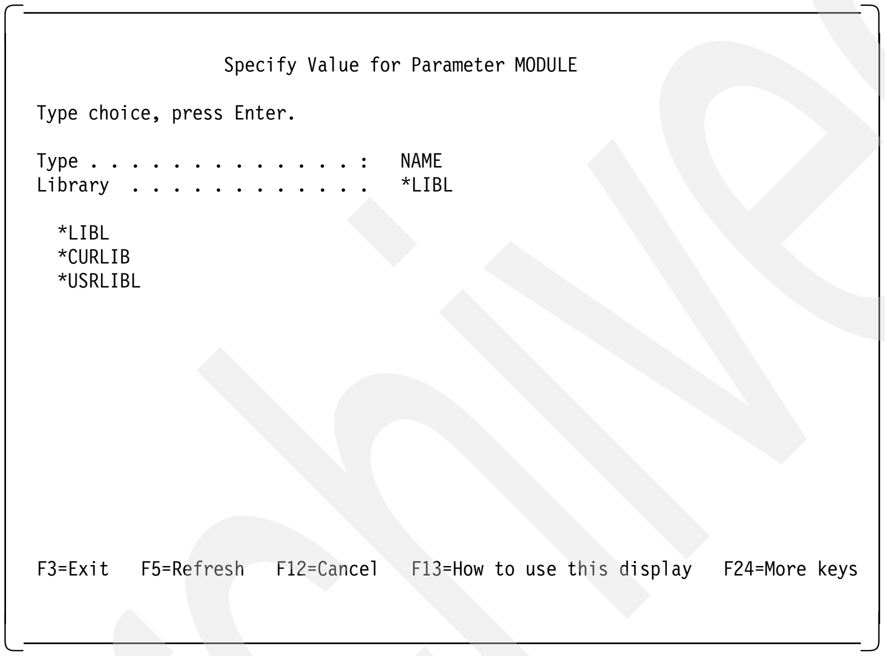
And if you want to know which value is suitable for particular field, then just type ″?″ on that field and press Enter.



```
                           Create Program (CRTPGM)

 Type choices, press Enter.

 Program  . . . . . . . . . . . .                         Name
   Library  . . . . . . . . . . .     *CURLIB     Name, *CURLIB
 Module . . . . . . . . . . . . .   *PGM          Name, generic*, *PGM, *ALL
   Library  . . . . . . . . . . .     ?           Name, *LIBL, *CURLIB...
               + for more values

 Text ′description′ . . . . . . .   *ENTMODTXT









                                                                       Bottom
 F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel
 F13=How to use this display        F24=More keys
```

Figure 7. Create Program Prompt Text

Then you can see the candidate values for that field. And you can choose one value from among these values.



```
                 Specify Value for Parameter MODULE

 Type choice, press Enter.

 Type . . . . . . . . . . . . . :   NAME
 Library  . . . . . . . . . . .     *LIBL

   *LIBL
   *CURLIB
   *USRLIBL











 F3=Exit   F5=Refresh   F12=Cancel   F13=How to use this display   F24=More keys
```

Figure 8. Specify Value for Parameter MODULE

To make it even easier (you might say even more annoyed, but remember those first days of your UNIX experiences: are you sure those days were necessary evil? Wouldn't you be glad if you could do without it?), all of the commands are provided by menu. These menus can be grouped by subjects, verbs, objects, and so on. If you are familiar with SMIT on AIX, you know the productivity difference between menu support and simple command line support. Consider that all of the AS/400 commands come up with a SMIT like interface: and even better. You cannot possibly get lost on the AS/400 system.

Enough said about that, we list the UNIX commands equivalents of the AS/400 commands first.

3.2.1.2 UNIX Commands Equivalents for Integrated File System Commands

You can use this subsection as a quick reference to find the AS/400 integrated file system commands for the function (UNIX commands equivalents on the AS/400 system, so to speak). Some integrated file system commands have an alias to help the users with a UNIX background. For example, the Changing Current Directory command is `CHGCURDIR` on the AS/400 system. Its alias is `cd`. Typing either `CHGCURDIR` or `cd` on the command line has the same effect: they both change the current directory.

Note: The AS/400 command interpreter is not case-sensitive. We just used lower case for the UNIX commands and upper case for the AS/400 commands. It is just because of the documentation convention; in most, if not all, AS/400 documentation, the commands are in upper case. Typing either `cd`, `CD`, `CHGCURDIR`, `chgcurdir`, `chdir`, `CHDIR`, or even `ChGcUrDiR`, changes the current directory.

`cd`

Change Current Directory. This changes the directory to be used as the current directory. The AS/400 integrated file system command for this is `CHGCURDIR`. It has an alias such as `cd` or `CHDIR`.

`chgrp`

Change Primary Group. This changes the primary group from one user to another. The AS/400 integrated file system command for this is `CHGPGP`.

`chmod`

Change Authority Value. This turns authority on or off for an object. The AS/400 integrated file system command for this is `CHGAUT`.

`chown`

Change Owner. This transfers object ownership from one user to another. The AS/400 integrated file system command for this is `CHGOWN`.

`cp`

Copy. This copies a single object or a group of objects. The AS/400 integrated file system command for this is `CPY`.

`ln`

Add Link. This adds a link between a directory and an object. The AS/400 integrated file system command for this is ADDLNK.

ls

Display Object Link. This shows a list of objects in a directory and provides options to display information about the objects. The AS/400 integrated file system command for this is DSPLNK.

ls -l

Display Authority. This shows a list of authorized users of an object and their authorities for the object. The AS/400 integrated file system command for this is DSPAUT.

mkdir

Create Directory. This adds a new directory to the system. The AS/400 integrated file system command for this is CRTDIR, MD, or MKDIR.

mv

Move or Rename. This moves an object to a different directory or changes the name of an object in a directory. The AS/400 integrated file system command for this is MOV, MOVE, RNM, or REN.

pwd

Display Current Directory. This shows the name of the current directory. The AS/400 integrated file system command for this is DSPCURDIR.

rm

Remove Link. This removes the link to an object. The AS/400 integrated file system command for this is RMVLNK, DEL, or ERASE.

rmdir

Remove Directory. This removes a directory from the system. The AS/400 integrated file system command for this is RMVDIR, RD, or RMDIR.

tar/cpio

Restore or Save. This copies an object or group of objects from a backup device to the system or from the system to a backup device. The AS/400 integrated file system command for this is RST, or SAV.

var=pwd

Retrieve Current Directory. This retrieves the name of current directory and puts it into a specified variable (used in CL program). The AS/400 integrated file system command for this is RTVCURDIR.

3.2.1.3 Integrated File System Unique Commands

The AS/400 integrated file system also has its unique commands. They are:

CHGAUT

Change Authority Value. This change a user′s authority for the object.

CHKIN

Check In. This checks in an object that was previously checked out.

CHKOUT

Checks Out. This checks out an object, which prevents other users from changing, renaming,or removing it.

CPYFRMSTMF

Copy From Stream File. This copies data from a stream file to a database file member.

CPYTOSTMF

Copy To Stream File. This copies data from a database file member to a stream file.

WRKAUT

Work With Authority. This shows a list of users and their authorities, and provides options for adding a user, changing a user authority, or removing a user.

WRKLNK

Work With Object Links. This shows a list of objects in a directory and provides options for performing actions on the objects.

WRKOBJOWN

Work With Objects by Owner. This shows a list of objects owned by a user profile and provides options for performing actions on the objects.

WRKOBJPGP

Work With Objects by Primary Group. This shows a list of objects controlled by a primary group and provides options for performing actions on the objects.

3.3 Integrated File System and Porting

This section discusses various topics related to UNIX C applications porting to the AS/400 system in terms of a file system: that is, the topics related to the integrated file system.

3.3.1 File Descriptor Management

File descriptors are non-negative integers that the integrated file system uses to identify the files being accessed by a particular process. Whenever the integrated file system opens an existing file, or creates a new file, it returns a file descriptor that we use when we want to read or write the file. Each file descriptor refers to an open file description, which contains information such as a file offset, status of the file, and access modes for the file.

On a UNIX system, when a process starts, three file descriptors are already opened, numbered 0 through 2 representing standard input, standard output, and standard error. It is defined as STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO. Unfortunately, there are no such file descriptors in the integrated file system. If a file is opened the first time in the job, the file descriptor may begin with 0, which is the standard input descriptor on a UNIX system.

Note: The file descriptors 0, 1, and 2 associated with STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO are not features of kernel. This is a convention employed by the UNIX shells and many UNIX applications. OS/400 does not support UNIX shells.

3.3.2 File Pointer and File Descriptor

File pointers are the implementations for input/output in C and their structures are defined in the headfile <stdio.h>. File pointers are used in C when a file is created or opened for reading or writing. Information about the file I/O is stored in this file pointer structure, such as the buffer pointer, some flags, counters and the file descriptor. Because the standard file I/O functions use the system functions, such as `open()`, `create()`, `read()`, and so on, the file pointers are mapped to the file descriptors. Table 11 shows the mapping of file pointers and file descriptors on the UNIX system.

Table 11. Mapping of Standard I/O Descriptors and Pointers in UNIX

File Descriptor	File Pointer
0	stdin
1	stdout
2	stderr
Non-negative integer	File pointer

ILE C/400 file pointers do not use file descriptors. However, ILE C/400 supports standard I/O file pointers, such as stdin, stdout, and stderr. They are simulated as files by the ILE C/400 library routines.

Therefore, the standard file I/O functions do not access stream files of the integrated file system directly. This is supported, though; that is, your applications with standard file I/O functions can access stream files of the integrated file system with the PTF SF20204 for OS/400 V3R1 which provides a new runtime for the integrated file system interface.

3.3.3 Data Conversion

When you access files through the integrated file system interface, data in the files may or may not be converted, depending on the conversion type requested when the file is opened. The file can be opened or transferred either in binary modes or in text modes. When the data is read or transferred from the file in binary modes, it is not converted. However, when the data is read or transferred in text modes, it is converted.

For true stream files, the national language support-specific characters, any line-formatting characters, such as carriage return, tab and end-of-file characters, are just converted from one code page to another.

When reading from record files that are being used as stream files, end-of-line characters (carriage return and line feed) are appended to the end of the data in each record. When writing to record files:

- End-of-line characters are removed.
- Tab characters are replaced by the appropriate number of blanks to the next tab position.
- Lines are padded with either blanks (for a source physical file member) or nulls (for a data physical file member) to the end of the record.

FTP: When a file is transferred in text modes using FTP, the data is converted as follows.

From the AS/400 system
The data is converted to ASCII code if the ASCII code is used on the other system. The data is converted from the national language support code page of OS/400.

To the AS/400 system
The data is converted to EBCDIC code if the ASCII code is used on the other system. The data is converted to the national language support code page of OS/400.

3.3.4 Code Pages

When the data is read from the file, it is converted from the code page of the file to the code page of the application, job, or system receiving the data. When the data is written to the file, it is converted from the code page of the application, job, or system to the code page of the file.

3.3.5 What Level of Portability Do I Have?

Most UNIX C applications should fall into one or a mixed combination of the following categories:

- ANSI C compliant
- POSIX.1 compliant
- POSIX.2 compliant
- Character I/O based
- XWindows based
- Others

Not all of these are supported on the AS/400 system. Remember we position the AS/400 system as a server-of-choice in the client/server paradigm. Our main interests are in:

- Modernizing the existing AS/400 applications and do it in UNIX way.
- Porting a relatively modernized version (that is, client/server based UNIX C applications) of its server portion to the AS/400 system.

Given that point, those applications with heavy use of either POSIX.2 compliant, character I/O based, or XWindows based can be quite costly in porting to the AS/400 system. Or they can wait until we further enhance the openness of the AS/400 system. They can be either old type applications where the data serving server portion and the display serving client portion of the code exists in the same module or basically the client side application.

On the positive side, for those compliant with ANSI C, porting them to the AS/400 system should experience only a few problems if any. Our ILE C/400 is fully ANSI C compliant and we have had that for a long time.

As you might have guessed, with OS/400 V3R1 and its new integrated file system, we entertain the second category group: those that are POSIX.1 compliant. We want to invite two groups here, basically. They are:

- Application developers who directly use POSIX file I/O APIs
- DB middleware vendors

These two groups who might have regarded the AS/400 system as an impractical choice for their platform for porting cost and performance of file I/O, now will find it more attractive with the new features of the integrated file system. The issue of the second group is discussed in the DB chapter. Suffice it to say here that they will now find the AS/400 enough to attempt porting their middleware to run on the AS/400 platform. We expect it will come and the rest is quite transparent from user′s and application developers/administrators′ standpoint.

3.4 Example Programs for Integrated File System

Appendix Appendix B, “Integrated File System Example Programs” on page 227 provides some example programs that might be helpful to understand or work with the integrated file system.

Chapter 4. Process Management

4.1 Introduction

Process Management is one of the areas where the differences between UNIX systems and the AS/400 operating system are most apparent. This chapter does not go into detail about what the major differences are, it just brings up the key concepts that are relevant for a programmer, and that plays an important part when porting software.

To reflect the efforts being done to follow existing open standards, a number of new process related concepts have been added to V3R1 of OS/400. Traditionally it has only been concerned about the concept of *jobs*. However, there are also a lot of things the UNIX implementations and OS/400 implementation have in common, both on an operating system level and on a language level.

It is important to remember that most of the functions mentioned in this chapter use header (include) files from the library QSYSINC, which is optionally installable. Make sure this library is installed on your system before using any of the functions. This can be done by using the command:

```
CHKOBJ OBJ(QSYSINC) OBJTYPE(*LIB)
```

Additionally if spawn(), wait(), waitpid(), or pipe() is to be used, the optionally installable library QCPA must exist on the system, even though threads are not used. This chapter provides information about:

- General process/job/thread information
- Signals
- Authority considerations

4.2 Processes

4.2.1 Processes in UNIX

In UNIX, every process has a unique non-negative process identifier. Since it is guaranteed to be unique at every point in time, that is, two processes cannot execute in parallel with the same identifier, it is often used by programs, together with other identifiers, to provide means for uniqueness. Some examples are message IDs in SMTP (Simple Mail Transfer Protocol) and newsgroup article IDs, but also the `tmpnam()` function uses the process identifier when generating a path name.

4.2.2 Processes on the AS/400 System

In the AS/400 operating system, a process does not correspond directly to the same concept in UNIX. The traditional OS/400 equivalent of a process has been the concept of a *job*, however with the addition of `threads` into the operating systems, the traditional way to look at that relationship has been slightly altered.

An AS/400 *job* has a unique qualified job name. The *qualified job name* consists of three parts: the job name (or simple job name), the user name and the job number. For interactive jobs, the *job name* is the same as the name of the workstation you signed on to. For batch jobs, it is usually possible to specify a customized job name. Note that this only applies to when the batch job is started using the SBMJOB (Submit Job) command. More information is available in 4.6, "Starting and Stopping Processes/Threads" on page 65. The job name can be up to 10 characters long.

The *user name* is the name of the user profile under which the job is started. The same concept is used in UNIX related operating systems. For interactive jobs, the OS/400 user name is the name entered on the sign-on display, and for batch jobs, it is usually possible to specify the user profile under which the job is to run provided one has sufficient authority.

The *job number* is a unique number assigned by the system. It is used to identify jobs, that have identical user names and job names associated with them. The job number consists of six numeric digits. The job number is what most closely resembles the UNIX concept of a process ID.

4.3 Threads

Threads is relatively a new concept. This book does not cover porting aspects of threaded applications, since it currently is a little bit too early to do so. There are not enough threaded applications on the market today to be discussed here. However, basic information about the threads implementation of OS/400 are covered, since it can be used as a means for porting process related functionality, which exists in most UNIX and POSIX compliant operating systems, but so far does not exist on the AS/400 system.

4.3.1 Threads in UNIX

Some UNIX operating systems or products are implementing *threads*. Threads are sometimes referred to as lightweight processes and provide a technique for concurrent programming by allowing multiple flows of processing within a process. Each thread in a process is a separate processing flow, using fewer system resources than a traditional process and are created with less system overhead. The same as traditional processes, threads can be run independently by the system.

Different threads within a process typically run the same code and are able to share the same data, including global storage, heap, and open files. In AIX 3.2, it was introduced with the DCE (Distributed Computing Environment) implementation, since OSF (Open Software Foundation) designed DCE to make use of a thread implementation known as .pthreads. If the operating system did not supply thread support, the DCE implementation itself had to provide it.

Pthreads are covered in the POSIX.1c standard. The POSIX.4 WG (Portable Operating System Interface.4 Work Group) has four major projects:

- POSIX.1c (was called POSIX.4a and deals with pthreads)
- POSIX.1d (renumbered from POSIX.4b)
- POSIX.1i
- POSIX.1j (renumbered from POSIX.4d)

Both the AIX 3.2 implementation provided with DCE and the OS/400 implementation are based upon a *draft* of the preceding standard. There are also other threads packages, such as Sun′s LWPs and Mach′s C-threads.

4.3.2 Threads in the AS/400 System

Threads in OS/400 are provided with a free, but separately orderable feature called CPA (Common Programming APIs). Aside from threads, CPA also supports the following:

- Thread synchronization features and wrappers to ensure a thread-safe and thread-enabled C runtime environment (with some exceptions, where the functions are using static variables, such as `ctime`()).
- Thread-enabled file I/O model, based on the POSIX standard 1003.1.
- Thread-enabled socket API, based on OS/400 sockets and BSD4.3 functionality.

Additionally, certain process control functions are provided with CPA, even though they are *not* thread-enabled. See 4.6, "Starting and Stopping Processes/Threads" on page 65 for details.

Basically, some UNIX systems such as AIX support thread-safe versions of these functions:

```
int    asctime_r(const struct tm *, char *, int);
int    ctime_r(const time_t *, char *, int);
int    gmtime_r(const time_t *, struct tm *);
int    localtime_r(const time_t *, struct tm *);
```

In AIX, you have to link to the thread-safe library (libc_r.a) to make use of these. The main difference is that you provide a pointer to allocated memory in the call.

In AIX, you must also define the _THREAD_SAFE symbol in order for the preprocessor to find the prototypes. In Sun Solaris, the equivalent symbol is _REENTRANT. On HP_UX 9.0.3, there are no thread-safe versions of gtime() and localtime(), however, there are two versions of ctime() and asctime() depending on the number of arguments used. These are:

```
char *nl_asctime(struct tm *, char *, int);
char *nl_ctime(long *, char *, int);
char *nl_ascxtime(struct tm *, char *);
char *nl_cxtime(long *, char *);
```

On HP_UX, use `#define _INCLUDE_HPUX_SOURCE` to make use of these functions. We could not find any equivalent function on SunOS.

In the OS/400 standard `<time.h>` include file, the four -r functions can be used without having to define any symbol. However, the `ctime_r()` and `asctime_r()`

calls have a different number of arguments compared to Sun Solaris and AIX. It appears that the ″length″ argument has been omitted, which makes them very similar to the HP_UX `nl_ascxtime()` and `nl_cxtime()` calls.

In the CPA threads implementation, multiple OS/400 jobs share the same program storage, or ILE activation group (see Chapter 6, “Development Environment on AS/400 System” on page 157 for information about ILE). In other words, a thread in OS/400 is an OS/400 job with its own job identifier. All threads sharing an ILE activation group are considered to form a process. The main differences between a threaded job and a regular job are:

- The sharing of static storage with other threads.
- The way the job is started as well as the job type.

There are three kinds of identifiers involved in this scenario:

Job Identifier	Qualified Job Name. Every separately-executable piece of processing in OS/400 *always* has a job identifier that is generated by the operating system. This includes a separate non-threaded job as well as the executable components of a process (that is, each thread in a threaded program).
Process Identifier	In V3R1 of OS/400, this is the job identifier of the main thread. If the process is non-threaded, it is identical to the job identifier. In V3R6 of OS/400, it is a separate identifier, totally independent of any externally available piece of job information.
	All jobs within a process share the same process identifier.
Thread Identifier	The job number of that thread + 1.000.000 for each time the tread is reused from the standby pool.
	Further information about OS/400 threads, linking as well as discussions about thread-safe and thread-enabled functionality and other thread related details is found in the *CPA Extensions for OS/400 Reference* (SC41-3820) manual.

4.4 Process Groups and Job Control

This section describes some considerations related to process groups and job control.

4.4.1 Process Groups and Job Control in UNIX

In addition to having a process ID, each process in a UNIX operating system that supports what is commonly known as *job control*, belongs to a process group. A process group is a collection of one or more processes. Each process group has a unique process group ID. The function getpgrp() returns the process group ID of the process issuing the call. The most important attribute of a process group is that it is possible to send a signal to every process in the group by just sending the signal to the process group leader. Each time one of the standard UNIX shells that supports job control creates a process to run an application, the process is placed into a new process group. When the application spawns new processes, these are members of the same process group as the parent.

Note here that not all UNIX operating systems supports the concept of job control and even if the operating system supported it, it is not certain that the shell used to start a job would support it.

Job control is a feature added by Berkely around 1980 and it allows the capability of determining which jobs can access the terminal and which jobs are to run in the background. POSIX.1 specifies that if the symbol _POSIX_JOB_CONTROL is defined in <unistd.h>, job control is supported.

OS/400 does not support job control as defined in POSIX.1. First, it does not have a controlling terminal in the traditional sense. Regardless of what is entered on the keyboard, no SIGINT, SIGTSTP, SIGCONT, or SIGQUIT is sent to the foreground process group. More information about signals is found in 4.5, "Signals" on page 55. Second, only one process can have access to the display and the keyboard and this cannot be altered using tcsetpgrp(), the UNIX way of specifying if a process group is to be placed in the foreground. Actually, neither tcsetpgrp() nor tgetpgrp are implemented into OS/400.

However, OS/400 allows the use of certain job control related functions, such as getpgrp() and in V3R6 of OS/400, setpgrp(). The latter function is usually used either by a parent process to set the process group of a child or by the child to set its own process group to ensure that it is different from the parent process. At the time of writing this redbook, setpgrp() is not supported in V3R1 of OS/400. However it is possible to specify the process group of a new spawn()ed process by using the SPAWN_SETGROUP option.

By default, when you start an OS/400 job, it implicitly has a process group ID identical to the job number. In V3R1 of OS/400, this identifier is also identical to the process ID, whereas in V3R6 of OS/400, it has no relation to the process ID whatsoever. The OS/400 resources representing the process group are, however, not explicitly created until the process has expressed some kind of interest in creating one. This way of implementation facilitates for traditionally developed software to run unaffected by process groups or job control.

The indicator causing the job to actually form a process group with itself as process group leader, can be a call to any of the process related APIs, such as `getpid()`, `spawn()`, `wait()`, and `sigaction()`. The full set of APIs that enable a process to:

- Create a process group, if it does not already exist.
- Add itself as an entry in the process table.
- Enable the process to receive signals (4.5, "Signals" on page 55).

are found in Figure 12 on page 59.

If the job uses `spawn()` to create a child process, the child inherits the process group ID from the parent unless the SPAWN_NEWPGROUP flag is specified in the call. This causes a new process group to be created for the child. The SPAWN_SETGROUP flag also allows the possibility of setting the process group ID of the child process to a specific value. If the process group number is not valid, an error occurs.

Most UNIX related operating systems also incorporate the concept of a *session.* This is a collection of process groups. Each process group is a member of a session. A newly-created process joins the session of its creator. The `setsid()` function is used to create a new session and the most common use of `setsid()` is to effectively disconnect from the controlling terminal. This is especially relevant when dealing with *daemon* processes or processes that do not require stdin input.

The `setsid()` function demands that the process it is called from must *not* be a process group leader. This means, that in order to ensure that the disconnect from the terminal is successful, the process must `fork()`, exit, and let the child do the `setsid()`. Another way is to make this within an `if` statement, where the process verifies that it is not a process group leader by comparing its process ID with its process group ID. This prevents a sometimes unnecessary fork().

```
/* fork first child */
if( ( childpid = fork() ) < 0 ) {    1
  /* exit with non-zero return code */
  exit( 1 );
}
else if( childpid > 0 ) {
  /* parent exits */
  exit( 0 );    /* we are the parent  */   2
}
/* else the child process continues */

/*
 * Set up as a server.   Disconnect from terminal.
 */
setsid();    3
```

Figure 9. Disconnecting from Terminal. This is how UNIX processes usually disconnect from a controlling terminal.

Notes:

1 The job fork()s to let the shell think the command is done. The child inherits the process group ID of the parent, but gets a new process ID. This means the child is not a process group leader and that makes it possible to perform the setsid() in **3**.

2 The parent process exits. If the fork() fails (return -1), the reason could be that the maximum number of processes in the system or for the current user has been reached. It exits with return code 1, otherwise it returns with code 0, indicating that the child process is now active.

3 We are now certain that setsid() should succeed, since the child is not a process group leader.

It is also possible to use getppid() (Get Parent Process ID) to see if the parent is *init*, in other words, the pid is 1. This means one of two things: Either the parent process has exited and let the *init* process take care of the child, or the program (if a Daemon) was started from an entry in the inittab file. In the latter case, there is no reason to call setsid(), since *init* is already disconnected from the controlling terminal and this is inherited by any child process created.

4.4.2 Process Groups and Job Control on AS/400 System

In OS/400, most of these steps are irrelevant. As mentioned earlier, the need for disconnecting from the terminal is not as accentuated as in UNIX, since there is no way the interactive process can receive any signals, other than explicitly sending them or by using `alarm()`.

Note: This is only true for ILE C runtime signals. OS/400 signals can be asynchronous. For details, refer to 4.5, "Signals" on page 55.) However, it *could* be worthwhile to examine it if the current process is an interactive process or a batch process and maybe even to some extent, simulate `setsid()` by making the interactive job a batch job.

There is no *init* process in OS/400, but the `getppid()` call (Get Parent Process ID) returns 1 anyway if no parent process exists. The concept of a parent process in OS/400 is further explained in 4.6, "Starting and Stopping Processes/Threads" on page 65, but suffice to say here, that there is *only one way* for a process to be considered a child or a parent and that is the use of the `spawn()` call.

As a direct effect of this, the `getppid()` call in OS/400 is return 1, regardless if it is an interactive (foreground) job, or if it has been started using the traditional method of starting a batch job in OS/400, the SBMJOB (Submit Job) command. This means, that if a UNIX program is using the `getppid()` call to see if *init* is the parent as previously described, and, if it is, refrain from making the job a background job. The code would not work properly on the AS/400 system.

Another way to look at this issue is to see what the program really is supposed to do. If the purpose of the code sample in Figure 9 on page 48 is only to verify that the job really runs in the background, OS/400 must use some other method of determining if the if the current process is an interactive process or a batch process. On the AS/400 operating system, there are basically three ways of accomplishing this:

Command Definition On the the AS/400 system, it is possible to create a *command*, that is able to call the actual program that can call a CPP (Command Processing Program). Besides the capabilities of letting developers supply help texts and parameter validation programs verifying that the parameters are sent in a proper sequence to the CPP, it also allows the developer to specify the environment the process is to run in. In other words, it is possible to specify that this program should be run interactively, or as a batch job.

Of course it is possible for a user to change this command attribute, which means a major inconvenience both for the developer and for the user if the CPP expected to run in another kind of environment.

RTVJOBA

One of the most common methods to verify if a job is running in a batch environment or interactive environment is to use the CL (Control Language) command RTVJOBA (Retrieve Job Attributes). From V3R1 of OS/400, it is possible to create ILE versions of CL programs, which means that it is perfectly valid to link such a program module to the main program. An example of such an attempt is shown in the following figure:

```
PGM        PARM(&JTYPE)
DCL        VAR(&JTYPE) TYPE(*CHAR) LEN(1)
RTVJOBA    TYPE(&JTYPE)
ENDPGM
```

Figure 10. RTVJOBA. A CL program is used to find out if the job is an interactive job or a batch job.

This program is called with one parameter, a pointer to a character. It returns 0 if the job is a batch job, or 1 if the job is interactive. The C program used to call it could have the following layout:

```
#include <stdio.h>

void TJOB2(char *a);

void main()
{
  char a;

  TJOB2(&a);

  if(a == '1')
    printf("This is an interactive job.");
  else
    printf("This is a batch job.");

  return;
}
```

Figure 11. Figuring Out Job Type. This program inquires from the program in Figure 10 about the type of the job.

APIs

The two previous examples have been a little bit limited in regard to functionality. The command method only sees to it that a program executes in its proper environment and the RTVJOBA method can only find out if the job is interactive or a batch job. The latter method supports the UNIX way quite well and would definitely do as a substitute for the getppid() call, but we can actually find out even more about the current job type by using operating system APIs such as QUSRJOBI (Retrieve Job Information) and QWCRJBST (Retrieve Job Status).

In OS/400, the batch job could have originated from different sources. The most common batch job types include:

- BCH (Batch) - batch jobs started with SBMJOB (Submit Job).
- BCI (Batch Immediate) - batch jobs started as threads or spawn()ed.
- EVK (Evoke) - SNA (Systems Networking Architecture) TP (Transaction Programs) that are evoked.

There are also some other batch statuses, but they are mostly related to the System/36 environment.

After we have performed the first steps in Figure 9 on page 48, the behavior of the UNIX program has to be translated to OS/400 concepts. If the job is interactive, the UNIX program `fork()`s a new copy of itself before it disconnects from the terminal. We can apply one of the job creation methods described in 4.6, "Starting and Stopping Processes/Threads" on page 65 and restart the current program in batch mode. In other words, it is not possible to change the type of the job from interactive to batch while it is running. The disconnect from the terminal is automatically done for a batch job and the only way it can communicate is by reading and writing from sources other than the terminal. If stdin/stdout is used in the program, such as a UNIX filter or something similar, these files are overridden before program invocation.

For example, if a job is to read stdin and write to stdout, but is submitted to batch, you can override these files to operating system database files by using the `OVRDBF` command. An example is `OVRDBF FILE(STDIN) TOFILE(INFILE)`. Note that stream files or files in file systems other than /QSYS.LIB file system are not allowed.

If any stdin, stdout, or stderr has not been overridden, the output is sent to the printer file QPRINT and the input is read from a file called QINLINE, which should exist in the library list. More information about these files is found in the *ILE C/400 Programmers Guide*, SC09-2069.

This method of restarting the program assumes, of course, that steps performed before the second program invocation can be repeated without causing unwanted effects.

To summarize: applications that make use of process group and session related calls discover that OS/400 differs to some extent in how it handles sessions and how a process detects if it is interactive or running in batch mode. Sessions are **not** supported and until session support is available on OS/400, the restriction that the process group assigned in the `setpgrp()` call must be within the session of the calling process will not be enforced.

There are some job control calls that are not supported on OS/400, or where OS/400 behavior is slightly different than suggested by POSIX.1. These are:

`setsid()`	If the purpose of the `setsid()` call is to make a process disconnect from the controlling terminal, it is redundant, since OS/400 does not have a controlling terminal in its strict sense. However, it is useful to find out if a job is interactive and if it is, make it a

batch job, and thereby loose all contact with the terminal. None of the standard I/O streams cause output to be read from or written from the terminal. A technique to perform this is described in this chapter.

Terminals and ttys Since OS/400 does not support the /dev file system, functionality referring to direct access to the terminal is usually not supported. These calls include:

- ctermid(), used to obtain the *terminal pathname* - usually in UNIX systems /dev/tty.
- tcgetattr() and tcsetattr(), used to save and set terminal attributes. Usually in System V environments, calls to ioctl() are made instead. The OS/400 ioctl() function does not support these requests.
- isatty(), used to verify that a file descriptor represents a terminal device.
- ttyname(), which returns the pathname of the terminal device that is open on a file descriptor.

getppid() Usually this call returns the parent process ID of the current process. In UNIX, if it returns 1, *init* is the parent process, which usually means that the child is not connected to a terminal, either because the parent has died, that it has perform a setid(), that the parent has disconnected from the terminal, or in some operating systems, that it has been started as a part of the inittab processing. In OS/400, it just means, that spawn() has not been used to create the job or that the parent has exited.

In additional to standard SVID and POSIX interfaces, OS/400 offers some extra job control support by introducing the following APIs:

QpOwChkPgrp() This function provides an AS/400 system a specific way to obtain the process table information for the members of a process group. Remember, that a certain process is not assigned to a process group or a position in the process table until it has indicated its interest in receiving some kind of signal or some other process related function. Information

	in the process table specifies information about current pid, the parent pid, and the process group of each of the processes as well as the current status of the process. This status indicates if the process has ended, and if it has been stopped by a SIGSTOP signal. Note that even though the effect of the HLDJOB command is very similar, it is *not* reflected in the job status. The status also indicates if the process is waiting for one or more child processes (wait() and waitpid()), or if a process has requested that the SIGCHLD signal should be generated for the process when one of its child processes is stopped by a signal.
QpOwChkPid()	QpOwChkPid() performs the same function as QpOwChkPgrp(), with the exception that it is called with a pid instead of a process group ID and only returns process table information for the specified pid.
QpOwGetPgrp()	Returns the process group ID of the calling process. It is equivalent to the getpgrp() call.
QpOwGetPid()	Returns the process ID of the calling process. It is equivalent to the getpid() call.
QpOwGetPidNoInit()	This function is identical to the getpid() and QpOwGetPid() calls except that it does *not* enable the process to receive signals. Functionally it is identical to the getpid(); QpOsDisableSignals(); combination of functions.
QpOwGetPPid()	The QpOwGetPPid() function returns the process ID for the parent process of the calling process. It is functionally identical to getppid().

4.5 Signals

4.5.1 Signals in UNIX

A signal is a way of handling asynchronous events. In other words, a program does not have to process any particular part of the program in order to receive a generated signal. Most of the time support for signals is implemented in the operating system, but at times it is also the runtime part of a language implementation. This is the case for most C and C++ implementations.

Signals have been provided since the early versions of UNIX, but the reliability has not always been astonishing and sometimes it is hard to turn off the selected signals when you are processing a critical region of the code. Additionally, the issue of compatibility between Berkely and AT&T implementations (especially when you consider the reliability extensions), was an open issue.

Signals are an integral part of multi-tasking in the UNIX/POSIX environment. They are usually used for a number of reasons, depending on the purpose of the programs that uses them. Examples of different areas where signal functions can come in handy are:

Maintenance Purposes Most UNIX systems send a signal to the process in the event of invalid pointers, or other indications of a bug in the program. Depending on how the signal handling is set up, this can cause a core dump to be generated and used for debugging purposes by the developers. Some developers consider the generation of a core dump as something that tells the users when something is wrong with the program and maybe gives the program a bad reputation. If this is the case, the signal (for example SIGSEGV or SIGFPE) is ignored.

Communication Events When two programs are communicating with each other over a descriptor, it could be a networking (IP) program, a pipe or something else, and the recipient side of the conversation terminates (normally or abnormally), the sending party receives a SIGPIPE event.

	This can help developers find problems in the logic of the code and can, of course, also be ignored by the developer. Other network communication related signals are SIGIO and SIGPOLL that indicate an asynchronous I/O event.
Timer Functionality	Either the alarm() function, or the timer related families setitimer(), or the POSIX.4 recommended timer_create() causes the signal SIGALRM to be generated.
User/tty Interrupts	Usually the interactive user can cause a signal to be generated by using certain key sequences. Typical examples are Ctrl-C, that generate SIGINT and, in most cases, Ctrl- \, which cause SIGQUIT. Other examples are Ctrl-Z, which cause the SIGTSTP signal to be sent to the process and, sometimes the non POSIX signal SIGINFO when using Ctrl-T.
IPC	A designer can let programs interact with one another by using signals as an IPC (Inter Process Communication) mechanism. In most cases, SIGUSR1 and SIGUSR2 are used for this purpose.
Process Tracking	A parent is notified when a child process has terminated by waiting for SIGCHLD.

There are numerous other things signals can help us with and these are only some of the most commonly used. Each process has an action to be taken in response to each signal defined by the system. During the time between the generation of a signal and the delivery of a signal (when the actual action is performed), the signal is said to be *pending*. In most cases, this state is determined by using the sigpending() function. It is also perfectly valid for the process to *block* it. If a signal that is blocked is generated for a process and the action for that signal is either the default action or to catch the signal, the signal remains pending for the process until the process either unblocks the signal or changes the action to ignore the signal.

Blocking is very useful if the program is in a critical section of the code. A signal can be specified to be blocked either in the sigaction() call (which is the POSIX way of implementing the Ansi C signal() call, and should, therefore, be preferred when coding a program), or in the sigprocmask() and sigsuspend() functions. Each process has a *signal mask* that defines the set of signals currently blocked from delivery and that is inherited by a child from its parent.

4.5.2 Signals in POSIX

POSIX defined in its 1003.1-1990 standard (also known as POSIX.1), 13 required signals. Six more were optional, but had to be implemented if the operating system defined the symbol _POSIX_JOB_CONTROL. If _POSIX_REALTIME_SIGNALS is specified in *unistd.h*, it means that the POSIX.4 additions to the POSIX.1 signals model have been implemented.

POSIX.1 relies to a great extent upon the Berkely signal standard with some improvements. The Berkely standard is, of course, a considerable improvement to the ANSI C signal standard, but is POSIX.1 really the ultimate goal?

POSIX.4 addresses some of the items considered as shortcomings of POSIX.1. Examples are:

- Lack of signals for application use. Basically, only SIGUSR1 and SIGUSR2 can be used.
- Lack of signal queuing.
- No signal delivery order.
- Poor information content. The recipient process knows little more than what signal has been delivered. Additional information must be provided using some other kind of (IPC) method.

4.5.3 Signals on the AS/400 System

Now that we have been dealing with signals in a couple of the sections in general, let us reach the key issue. To what extent does OS/400 support signals? We look at what ILE C/400 provides us with and add the extensions provided in V3R1 of OS/400.

4.5.3.1 ILE C/400

Well, we can go the easy way first and talk about ANSI C signals. These are defined as SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM. Standard C does not require that any of these signals are generated. An illegal memory reference may, or may not, generate a SIGSEGV. ILE C/400 supports all of these with the additions of :

SIGIO	Originally used for record file error condition, but with V3R1 of OS/400, the use of the fcntl() command F_SETOWN or the ioctl() request of FIOSETOWN is also possible to use with descriptors.

SIGOTHER	All *ESCAPE and *STATUS messages that do not map to any other signals.
SIGUSR1, SIGUSR2	

Additionally, SIGALL is an ILE C/400 extension, which allows users to register their own default handling function for all signals whose action handler is SIG_DFL.

ANSI C also specifies signal() to be used for specifying signal handlers and this function is supported by ILE C/400. Note, however, that support for sigaction() is included in V3R1 of OS/400 and V3R6 of OS/400. AS/400 system exceptions are mapped to C signals by the ILE C/400 runtime. You cannot register a signal handler in an activation group that is different from the one you want to invoke it from. The concept of activation groups is further explained in Chapter 6, "Development Environment on AS/400 System" on page 157. If a signal handler is in a different activation group from the occurrence of the signal it is handling, the behavior is undefined.

Signals can be raised implicitly or explicitly. To *explicitly* raise a signal, the raise() function can be used. Signals are *implicitly* raised when an exception occurs. Note that this only applies to the use of ILE C/400 signals, that is, the signal() function. See 4.5.3.2, "ANSI C, POSIX Integration" on page 60 for details. The *signal.h* header file contains a number of function prototypes associated with signal handling. In order to ensure that your program has access to *all* signal functionality, verify that your include statements mention #include <sys/signal.h>. If this is done, the compiler finds QSYSINC/SYS.SIGNAL, which in turn includes the ILE C/400 QCLE/H.SIGNAL, which only contains the C functionality. Note that the functions included in OS/400 POSIX signal functionality *requires* QSYSINC/SYS.SIGNAL to exist on the system. However, QSYSINC is optionally installable. Make sure QSYSINC is installed on your system before using any of the functions.

As mentioned in 4.4, "Process Groups and Job Control" on page 46, it is not possible to generate any of the commonly accepted interactive signals (SIGINT, SIGTSTP, SIGQUIT , or SIGINFO) from the keyboard. SIGTSTP is not even supported. Instead, the *only* way of causing one of these signals to be generated is to use either the raise() function provided in ILE C/400, or the kill() function provided in OS/400 V3R1. In fact, the *only* way to cause a signal to be generated that does not include any of these calls or an exception from the operating system, is to use the alarm().

In OS/400, a process is by default *not* eligible to receive signals from other processes or the operating system. The QpOsEnableSignals() function allows the calling process to receive signals from other processes or the system without having to call other signal functions that enables the process for signals.

The activities of explicitly creating a process group, adding a process to the process table, and enabling a process to receive signals are very closely related and usually performed in one step. In other words, if the process is enabled for signals, it is automatically added to the process table. In addition to the QpOsEnableSignals() call, this kind of process enabling is acquired if:

1. The job is a child process, that is, it was created using spawn().
2. The process is a parent process, that is, it is using spawn().
3. Any of the following functions are called:

- alarm()
- getpgrp()
- getpid()
- pause()
- QpOwGetPgrp()
- QpOwGetPid()
- sigaction()
- sigprocmask()
- sigsuspend()
- sleep()

Figure 12. Functions to Process Enable a Program. Any of these functions can cause a process group to be explicitly formed with the current process as the process group leader, add the process to the process table, and enable the process to receive signals.

If none of these conditions are met, the process is not enabled to receive signals. If another process tries to generate a signal to the process anyway, an error condition occurs.

The process to receive the signal is identified by a *process ID*. 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; see 4.4, “Process Groups and Job Control” on page 46 for more details). The process sending a signal must have the appropriate authority to the receiving process. The parent process is allowed to send a signal to a child process (see 4.6, “Starting and Stopping Processes/Threads” on page 65 for details). A child process is allowed to send a signal to its parent. A process can send a signal to another process if the sending process has *JOBCTL authority defined for the current process user profile or in an adopted user profile. Otherwise, the real or effective user ID of the sending process must match the real or effective user ID of the receiving process. An error condition results if the process does not have authority to send the signal to a receiving process. We discuss authority issues later in 4.6.5, “Process Authorization” on page 85.

There are some additional differences between how signal functionality can be used in OS/400 and the standard UNIX (POSIX, X/OPEN) model.

4.5.3.2 ANSI C, POSIX Integration

In most UNIX systems, the standard C signal functions, signal() and raise() can be used interchangeably with the POSIX calls sigaction(), and kill(). If one process indicates its interest in receiving, for example, SIGUSR1 by using signal(SIGUSR1, handler), it is perfectly valid for another process to send that signal to the first process using kill(<pid>, SIGUSR1). In a similar fashion, it is possible to use the following flow of events:

```
struct sigaction sig_parms;

sig_parms.sa_handler = handler;
sigemptyset(&sig_parms.sa_mask);
sig_parms.sa_flags = 0;
sigaction(SIGUSR1, &sig_parms, NULL);

raise(SIGUSR1);
```

This is not possible in OS/400, however. Generally it can be expressed as functionality belonging to two different families. The ILE C/400 runtime supports the signal() and raise() functions, whereas the operating systems supports sigaction() and kill(), and they *cannot* be used together. Either you are using the ILE C/400 flow of events or the functionality of the operating system.

4.5.3.3 Scope of Signal Information

On most UNIX systems, a process consists of a single flow of control. When the program in control needs to perform a task that is contained in another program, the program uses fork() and exec() to start a child process that is used to start the other program. The signal controls for the child process are inherited from the parent process. Changes to the signal controls in either the parent or the child process are isolated to the process in which the change is made.

In OS/400, when a program needs to perform a task that is contained in another program, there are a number of ways to accomplish this. If the other program is to run concurrently with the current program, it is possible to use SBMJOB or spawn(). When you use SBMJOB, nothing is inherited, not the current signal mask nor the signal actions. When you use spawn(), it is possible to specify if the signal mask of the parent is to be inherited, or if it is going to be set in the call. Additionally it is possible to specify if you want the child to inherit the parents ignore (SIG_IGN) or default (SIG_DFL) actions.

If the called program is *not* going to run concurrently, but instead have the functionality of a UNIX fork(); wait(), that is how the ANSI C system() call is usually implemented, the target program is run using the same process structure. As a result of this call and return mechanism, if a called program changes the process signal controls and does not restore the original signal controls when returning to its caller, the changed process signal controls remain in effect. The called program inherits the signal controls of its call, however:

1. The set of pending signals is not cleared.
2. Alarms are not reset.
3. Signals set to be caught are not reset to the default action.

Programs that use signals and change the signal controls of the process should restore the old actions or signal blocking mask (or both) when they return to their callers. Programs using signals should explicitly enable the process for signals when the programs begin. If the process was not enabled for signals when the program was called, the program should also disable signals when it returns to the process. In other words, if the first program is not enabled to receive signals, but it is a part of the program it calls, the target program should use QpOsDisableSignals() before returning to the caller.

4.5.3.4 Error Handling in a Signal Handler

On UNIX systems, an unhandled error condition in a signal handler usually results in the interrupt of the process. In OS/400, however, unhandled error conditions in the signal handler are implicitly handled. The signal handler is ended and the receiving program resumes running at the point at which it was interrupted. The error condition may be logged in the job log. Aside from the job log entry for the error, no further error notification takes place.

4.5.3.5 Termination Action

The OS/400 offers two types of termination actions. The termination action applied to most signals is to end the most recent request. This usually results in ending the current program, which is expected by most UNIX programmers. The second termination action is to end the process, which is more severe. The only signal with this action is SIGKILL.

4.5.3.6 Default Actions

In OS/400, some default actions for signals are different than on typical UNIX systems. For example, the OS/400 default action for the SIGPIPE signal is to ignore the signal.

4.5.3.7 Supported Signals and Signal Numbers

OS/400 does not implement all POSIX required signals; SIGHUP has been left out. A number of programs, however, are designed to install a signal handler for SIGHUP, which typically lets the programs reread their configuration files and reinitialize according to the (new) information.

This does not necessarily have to impose a problem, since, even though the signal is not sent, (OS/400 does not support the full concept of a controlling terminal) it is still possible to send the signal using raise() or kill().

The excerpt from <sys/signal.sys> in the following example indicates which signals are supported and those that are not in addition to their corresponding signal numbers:

```
/*-------------------------------------------------------------------*/
/*                    ANSI-C Required Signals                        */
/*-------------------------------------------------------------------*/
/*
 * #define SIGABRT     1    * Abnormal termination *
 * #define SIGFPE      2    * Erroneous arithmetic operation *
 * #define SIGILL      3    * Invalid hardware instruction *
 * #define SIGINT      4    * Interactive attention signal *
 * #define SIGSEGV     5    * Invalid memory reference *
 * #define SIGTERM     6    * Termination signal *
 */
```

```
/*-------------------------------------------------------------------*/
/*                     SAA or Extended Signals                       */
/*-------------------------------------------------------------------*/

#define SIGUSR1     7       /* Application defined signal 1 */
#define SIGUSR2     8       /* Application defined signal 2 */
#define SIGIO       9       /* I/O possible, or completed */
/*
 * #define SIGALL      10
 * #define SIGOTHER    11
 */


/*-------------------------------------------------------------------*/
/*                 Additional POSIX Required Signals                 */
/*-------------------------------------------------------------------*/

#define SIGALRM    14       /* Timeout signal */
#define SIGKILL    12       /* Termination signal (cannot be
caught,ignored) */
#define SIGPIPE    13       /* Write on a pipe with no readers */
#define SIGQUIT    16       /* Interactive termination signal */


/*-------------------------------------------------------------------*/
/*                 Additional POSIX Required Signals                 */
/*-------------------------------------------------------------------*/
/*  NOTE:  These signals are provided as an aid to application       */
/*         porting.  These signals are not generated by OS/400.      */
/*-------------------------------------------------------------------*/

#define SIGHUP     15       /* Hangup detected on controlling terminal
*/


/*-------------------------------------------------------------------*/
/*                       Job Control Signals                         */
/*-------------------------------------------------------------------*/

#define SIGCHLD    20       /* Child process terminated or stopped */
#define SIGCONT    19       /* Continue if stopped */
#define SIGSTOP    17       /* Stop signal (cannot be caught or
ignored) */


/*-------------------------------------------------------------------*/
/*                       Job Control Signals                         */
/*-------------------------------------------------------------------*/
/*  NOTE:  These signals are provided as an aid to application       */
/*         porting.  These signals are not generated by OS/400.      */
/*-------------------------------------------------------------------*/

#define SIGTSTP    18       /* Interactive stop signal */
#define SIGTTIN    21       /* Background read from controlling
terminal */
```

```
#define SIGTTOU    22        /* Background write to controlling
terminal */


/*-------------------------------------------------------------------*/
/*                   Additional X/Open 1170 Signals                  */
/*-------------------------------------------------------------------*/

#define SIGURG     23        /* High bandwidth data is available at a
socket */
#define SIGPOLL    24        /* Pollable event */


/*-------------------------------------------------------------------*/
/*                   Additional X/Open 1170 Signals                  */
/*-------------------------------------------------------------------*/
/*  NOTE:  These signals are provided as an aid to application       */
/*         porting.  These signals are not generated by OS/400.      */
/*-------------------------------------------------------------------*/

#define SIGBUS     32        /* Bus error (specification exception) */
#define SIGDANGER  33        /* system crash imminent */
#define SIGPRE     34        /* programming exception */
#define SIGSYS     35        /* Bad system call */
#define SIGTRAP    36        /* Trace/Breakpoint trap */
#define SIGPROF    37        /* Profiling timer expired */
#define SIGVTALRM  38        /* Virtual timer expired */
#define SIGXCPU    39        /* CPU time limit exceeded */
#define SIGXFSZ    40        /* File size limit exceeded*/


/*-------------------------------------------------------------------*/
/*              Signal names supplied for compatibility              */
/*-------------------------------------------------------------------*/

#define SIGIOINT   SIGURG    /* printer to backend error signal */
#define SIGAIO     SIGIO     /* base lan i/o */
#define SIGPTY     SIGIO     /* pty i/o */
#define SIGIOT     SIGABRT   /* abort (terminate) process */
#define SIGCLD     SIGCHLD   /* old death of child signal */
#define SIGLOST    SIGABRT   /* old BSD signal */
```

Figure 13. Signal.h. Supported and unsupported signals in OS/400.

The marked signal *#define*s are functions that are implemented in ILE C/400. These are documented in <signal.h>, which is included in the header file in Figure 13.

What can be noted is that the signal numbers are not identical to most UNIX systems. For example, it is easy to grow accustomed to the fact that SIGHUP is 1 and that kill -1 <inetd-pid> should refresh the inetd daemon. I also think that one of the things most UNIX students learn very quickly is to end jobs that do not exit normally using kill -9 <pid>. Suddenly, when you kill

-1, you send a SIGABRT, which is percolated to the next exception handler and when you send -9, suddenly the process receives SIGIO, which can cause it to take incorrect actions. The best way to prevent this is to use signal names instead of just numbers whenever possible. This could also facilitate porting to other platforms with different kinds of signal handling.

In order to be able to easily compare the AS/400 operating system signal numbers with the UNIX equivalents, the following figure shows the signal-number mapping in AIX:

```
 1) HUP       14) ALRM      27) MSG
 2) INT       15) TERM      28) WINCH
 3) QUIT      16) URG       29) PWR
 4) ILL       17) STOP      30) USR1
 5) TRAP      18) TSTP      31) USR2
 6) LOST      19) CONT      32) PROF
 7) EMT       20) CHLD      33) DANGER
 8) FPE       21) TTIN      34) VTALRM
 9) KILL      22) TTOU      35) MIGRATE
10) BUS       23) IO        36) PRE
11) SEGV      24) XCPU
12) SYS       25) XFSZ
13) PIPE
```

Figure 14. Signal Number Mapping in AIX

4.6 Starting and Stopping Processes/Threads

This section describes the topics related to starting and stopping of the processes and threads.

4.6.1 Arguments and Environment Variables

The reason we are talking about environment variables and argument lists when we are supposed to analyze different ways of starting, stopping, and verifying processing units of code, is, of course, that they play a major part when invoking applications. When coding a C program, one of the first things to learn is that the argument counter and argument vector is passed to the `main()` function when the program is started. Additionally, many UNIX programs (also DOS, OS/2, and VMS... programs) are depending heavily on environment variables and some have even based their entire configuration procedure on the existence of these.

4.6.1.1 Argument Lists on the AS/400 System

Argument lists work very much the same as in a UNIX environment. The main difference is that the calls to the programs are not performed in the same way from the command line. Either your program can have a parsing command, which takes care of oddities in regard to parameter parsing, or you can call your program directly using the syntax:

```
CALL PGM(foo) parm(bar 'baz' 124 '32')
```

Note that:

- String literals are passed with a null character.
- Numeric constants are passed as packed decimal digits. This means that the preceding argument 124 is not passed in as a null terminated string. Instead, it is passed as a packed decimal. If the UNIX environment is to be emulated, the number must be passed in quoted notation, such as in '32' in the preceding example.
- Characters not enclosed in single quotation marks are folded to uppercase. In the preceding example, `bar` is passed as `BAR`. However, `baz` is passed in its proper (lower) case.

In the C program, however, arguments are handled the same way as they are in most C implementations.

4.6.1.2 Environment Variables in UNIX

OS/400 implements environment variables and passes the proper argument counter and vector as a part of invoking a program. However, some differences compared to System V, BSD, and POSIX standards can be noted and that is what we are going to mention in this chapter.

Environment variables are character strings of the form "name=value" that are usually stored in an environment space outside of the program. Each program is passed an environment list. Similar to the argument list, the environment list is an array of character pointers with each pointer containing the address of a null-terminated string. The address of the array of pointers is contained in the global variable `environ`.

Historically, most UNIX systems have provided a third argument to the `main()` function, which provides the address of the environment list. However, ANSI C specifies that the `main()` function is to be written using only the argument counter and the argument vector. POSIX.1 specifies that the external variable `environ` should be used instead of the third argument. POSIX.1 also specifies that to access an environment variable, either the `getenv()` function or the environ variable is to be used, depending on how the environment is to be accessed. The POSIX.1 supplement also proposed the adding of the putenv() and clearenv() functions to add or change an environment variable or to clear the whole environment space. None of these functions suggest an easy way to remove one single variable. However, that is the purpose of the BSD 4.3 function unsetenv(), which is not mentioned in either XPG3, POSIX, or ANSI C. Additionally, BSD 4.3 suggests the use of setenv(), which works similar to putenv(), but makes it possible to separate the name of the variable and the value. In addition, it makes it possible to specify what is to happen if the environment variable is already set.

There are a number of pitfalls that applications using the proposed version of putenv() must take into consideration. In UNIX systems, one usually has to declare static storage with the name=value string and then pass the pointer to the static storage to putenv(). One common mistake is usually to allocate automatic storage before calling putenv() with potentially devastating results when the program later tries to reference a memory area that does no longer exist. One way of being able to handle this situation usually is to call the strdup() function to allocate the memory needed. Both putenv() and clearenv() are free to change the value of environ, which means that if a user initially makes a copy of the value, it might not be valid after a call to one of these functions.

The Rational in the POSIX.1 standard states that the two latter calls are being considered for an amendment to POSIX.1.

4.6.1.3 Environment Variables on the AS/400 System

So....how does OS/400 handle environment variables? Basically, the strings are stored in a temporary space associated with the job. If OS/400 V3R6 is used, users additionally have access to a number of commands that offer the possibility to manipulate environment variables from CL programs and that can be used from the command line. These commands do *not* exist in V3R1 of the operating system and at the time of writing this redbook, there were no plans to supply them as a PTF. The commands concerned are :

ADDENVVAR	Add Environment Variable - this command can be used to set an environment variable for a job. It is similar to the csh setenv function or the ksh, bsh export call.
CHGENVVAR	Change Environment Variable.
WRKENVVAR	Work with Environment Variables - this command allows the user to display or change environment variables for a job.

Even though V3R1 does not support these commands, it supports the functionality to create them by using the environment management functions mentioned initially. We investigated how this can be done, but first some information about differences in functionality in regard to most UNIX implementations.

If a shell script in a UNIX environment is to set environment variables, the call to it must usually look like `. ./envscript`. The reason for the initial dot (.) is that otherwise a subshell is invoked and the variable added. Unfortunately though, it is not accessible from the shell making the request, since it was allocated in the environment space of another process.

The AS/400 system, though, does not care very much about subshells. If a program, which essentially only makes a `putenv()`, is created, the variable is fully recognized from any program within the job. This also applies to processes, since OS/400 threads inherits the environment of the main thread.

There is a limitation of the number of environment variables allowed. The V3R6 limit is 1024 per process. After environment variables are set, they exist for the duration of the job. There is no way to remove an environment variable. However, the value can be set to `NULL` by using a subsequent call to `putenv()` specifying a value of NULL.

There is no default set of environment variables provided when a job starts. This also applies to the POSIX defined variables `HOME`, `LANG`, `LOGNAME`, `PATH`, and `TERM` as well as to `TZ` and the `LC...` variables used for locale information. Initially, when a job is started on OS/400, the `environ` variable is set to `NULL`, that is, it is not initialized and it will not be initialized regardless if an environment has been added previously in the program. On the AS/400 system, `environ` is initialized by a call to the functions such as `QpOzInitEnv()`, `putenv()`, `getenv()`, `QpOzGetEnv()`, or `QpOzPutEnv()`.

4.6.1.4 Inheritance

On a UNIX system, the `fork()` function creates a new process and extends the environment variables of the original process to the new process. Although OS/400 has no `fork()` function, environment variables *are* extended to a new job created when using the Submit Job (SBMJOB) command, provided any environment variables existed in the first job (this is available in V3R6 of OS/400 only). Additionally, it is possible to explicitly inherit environment variables using `spawn()` to create a new process, either by providing the `environment` variable in the call or by simply emulating the concept of a pointer to an array of character pointers (char **) and sending it into the spawn() call. Both spawn() and SBMJOB allows for the passing of arguments.

4.6.1.5 OS/400 Specifics

In addition to getenv() and putenv(), the AS/400 system also offers some extra environment related calls that perform basically the same function, but with the extra attribute of a CCSID (coded character set identifier) field. This allows the user to associate a CCSID with the environment variable other than what is currently used in the job. The functions are Qp0zGetEnv(), Get Value of Environment Variable (Extended), and Qp0zPutEnv(). These, in addition to Qp0ZInitEnv() referred to earlier, provide some OS/400 specifics in regard to environment variable handling.

4.6.1.6 V3R1 Environment Access

As mentioned previously, V3R1 of OS/400 does not allow for the use of the environment variable related commands. However, it is relatively easy for a user to implement such a concept. There are basically only three things that can seem a little different in comparison to most UNIX systems:

Qp0zInitEnv()	As mentioned previously, this call initializes the environ pointer. Note that it is only necessary to use it when this pointer has to be used explicitly. It is not necessary if access is only to be performed using getenv().
putenv()	You do *not* have to allocate static storage using the static identifier, malloc(), calloc(), or strdup(). The value is copied to the process related persistent data area and can be accessed from any program (or even outside a user program) as soon as a value has been assigned.
Header Files	In most UNIX implementations, the putenv() function prototype can be found in <stdlib.h>. This is also the case in V3R6 of OS/400, however in V3R1 of OS/400, it is necessary to include the AS/400 specific include file <qp0z1170.h>. This is also necessary if any of the OS/400 specific environment handling functions are to be used. Additionally, even though it is possible to create a program only using getenv(), using only <stdlib.h>, and without using the service program QP0ZCPA, this implementation of getenv() does *not* work with the

OS/400.. implementation of environment variables, since the vanilla version really is a part of ILE C/400 runtime. In other words, when using environment variable APIs in V3R1 of OS/400, use the <qp0z1170.h> header file.

```
#include <qp0z1170.h>
#include <stdio.h>

void main(int argc, char **argv)
{
  if (argc != 2)
  {
    fprintf(stderr, "%s: Error in syntax.\n", argv[0]);
    return;
  }

  if (putenv(argv[1]) < 0)
  {
    fprintf(stderr, "%s: Putenv failed.\n", argv[0]);
    return;
  }

  return;
}
```

Figure 15. ADDVAR. This is a small example of how ADDENVVAR *can be implemented in V3R1 of OS/400. It is to be called using* CALL ADDVAR PARM('name=value')*.*

```
#include <qp0z1170.h>
#include <stdio.h>

void main(int argc, char **argv)
{
  char *a;

  if (argc != 2)
  {
    fprintf(stderr, "%s: Error in syntax.\n", argv[0]);
    return;
  }

  if (!(a = getenv(argv[1])))
    fprintf(stderr, "%s: Getenv failed.\n", argv[0]);
  else
    printf("%s=%s\n", argv[1], a);

  return;
}
```

Figure 16. DSPVAR. This small sample shows how it is possible to examine a value of an environment variable. It is to be called using `CALL DSPVAR PARM('name')`.

```
#include <qp0z1170.h>
#include <stdio.h>
#include <string.h>

void main(int argc, char **argv)
{
  extern char **environ;
  int        a = 0;

  if (argc != 1)
  {
    fprintf(stderr, "%s: Error in syntax.\n", argv[0]);
    return;
  }

  if (Qp0zInitEnv() < 0)
  {
    perror("Qp0ZInitEnv");
    fprintf(stderr, "%s: Initenv failed.\n", argv[0]);
  }
  else
    for (a=0; environ[a]; a++)
      printf("%s\n", environ[a]);

  return;
}
```

Figure 17. ENV. This program performs something similar to the UNIX env *or DOS/OS/2* set*. It displays all environment variables by traversing the external* environ *variable after first initializing it.*

4.6.2 Threads and Spawning New Jobs

The following sections are going to bring up the concept of threads and spawning new jobs using spawn(). Both of these concepts involve initiating separate flows of control and both provide different methods of accomplishing tasks such as I/O and inheritance.

But the two methods also have some things in common. New jobs that are created are, for example, of the type BCI (Batch Immediate) using both interfaces. This particular job type is not taken into consideration by the subsystem the jobs are running in regard to the job queue entry specifying how many active jobs can be active from a certain job queue at a given time.

This means, that if you, for example, submit a threaded job using the job queue QBATCH in the QBATCH subsystem, the job queue takes this job entry into consideration. However, it does *not* recognize any thread created by the program in any way. Threads does not use the job queue of the job, and for this reason, it is possible to see hundreds of jobs being active in QBATCH, even though the subsystem has been carefully tailored not to allow more than a dozen.

The same concept applies to `spawn()` jobs. These are not threads, but regular OS/400 jobs. However, they do have the same job type, `BCI`, and an arbitrary number of these jobs can run in a subsystem.

4.6.3 Threads

This chapter is not going to consider porting between a UNIX threaded environment, such as the AIX implementation of *pthreads* or the POSIX.1c (previously POSIX.4a) standard. Instead, it focuses on the feasibility of using threads as a means of compensating the AS/400 system's lack of `fork()`ing ability. This issue has been ventilated thoroughly in this book, not the least of its practical impact in a networking environment. What we mention here is just basic issues about what you have to keep in mind when considering a thread-based implementation as a part of the porting work.

Two concepts are very important when dealing with threads are:

Thread-safe	This is a program or function that does not affect another thread, for example, by using the same static storage, creating IPC resources using a hard coded IPC key, or any other shared resource. Functions such as `ctime()` are *not* considered thread-safe.
Thread-enabled	This means essentially, that a resource can be shared among threads as if the access was performed in a single flow of control environment. One example is different threads reading from a file. When thread number 1 has read from the file, thread number 2 can continue and read from where the file pointer is located. In this case, however, the file is considered a resource and the access might have to be serialized depending on the circumstances. A thread-enabled program or function must also be thread-safe. It must provide extra logic to allow the program's functions and resources to be addressed and shared across multiple threads in a consistent manner.

Threads are part of a product called CPA - Common Programming APIs Toolkit/400. CPA is an optional, separately orderable, but free of charge part of the operating system, which means that no external prerequisites, such as some kind of runtimes, are needed if a threaded program is shipped to a customer. However, CPA *must* be installed both on the development system and on the system where the program is to run. Additionally the CPA library, QCPA, must be in the library list both when compiling (because of header files supplied in QCPA) as well as when linking and running the program because of service programs (*SRVPGM) that are supplied in the library.

In its current implementation, CPA can best be described as an early version of a finished product. It is not fully integrated into OS/400; that is, it does not have some of the usability characteristics IBM intends for the final version.

As the POSIX.1c (previously 1003.4a) draft standard for threads is approved, IBM will change CPA to be consistent with the standard. These changes may be incompatible with the current release and can require changes to programs using CPA functions in the future. As a CPA program developer, it is essential to be aware that the code might not work the same way (or at all) in future releases of CPA.

4.6.4 Jobs

The scope of this section is to bring up what methods there are, and the rules for:

- Calling other programs.
- Creating separate flows of control.

4.6.4.1 System()

In C, the traditional way of invoking a secondary program, wait for it until it has terminated, and resume processing has been to use `system()`. `system()` is required by standard ANSI C. It is not a part of the POSIX.1 standard, because it is not an interface to the operating system, but really an interface to a shell or a command interpreter. Instead, the POSIX.2 standard provides several hundred pages of documentation on the arguments to system.

Even though `system()` by itself usually is portable between ANSI C environments, its argument (the command string) is not. The string is heavily dependent on the interpreter, which in turn can be dependent on the operating system. This means that a C program in a UNIX environment, which makes the call `system("date > file")`, has to be changed in an OS/400 environment. Other interesting porting issues when dealing with `system()` is its exit status. In most UNIX flavors, the only way to cause a new process to start is to internally use fork(), exec(), and waitpid(). Actually, this is sometimes the only alternative. As a result of this, the UNIX system() implementation is also built on these three functions. The direct effect is how the return codes are handled. Under UNIX:

- If either the fork() or waitpid() returns an error other than EINTR (interrupted system call), system() returns -1 with errno set to indicate the error.
- If the exec() fails, which usually means that something is wrong with the command or permissions, the return value is 127.
- If all three calls succeed, the return value from system() is the termination status of the shell in the format specified for waitpid().

An interesting issue here is the waitpid() return format. waitpid() can be passed a pointer to an integer, where it returns the termination status of the process. POSIX.1 specifies that this status is to be looked at using various macros defined in <sys/wait.h>. The three main mutually exclusive macros are WIFEXITED, which is true if the child has terminated normally; WIFSIGNALED, which is true if the child has terminated abnormally (usually by a signal); or WIFSTOPPED, which is true if the child is currently stopped (through SIGSTOP or SIGTSTP). Since this status information is used by system(), there are programs that rely upon the return code from system() to derive the reason why the process ended and determine if the process ended successfully.

This procedure does *not* work under OS/400 since neither fork(), exec() or waitpid() are used. Instead, the returned value, when invoked with a proper command string is 1 if system() fails and 0 if it succeeds. Additionally, it returns -1 if passed a NULL pointer as the command string and no attempt is done to invoke a command.

The differences in regard to return values between UNIX and ILE C/400 are shown in the following table:

Table 12 (Page 1 of 2). Behavior of the System() Function

Input	**Result**	**UNIX rc**	**ILE C/400 rc**
NULL	N/A	1	-1
Command could not be run	N/A	127	1
Valid command	OK	0	0

Table 12 (Page 2 of 2). Behavior of the System() Function

Input	Result	UNIX rc	ILE C/400 rc
Valid command	fork() fails	-1	N/A
Valid command	waitpid() fails	-1	N/A
Valid command	Signal delivered	A value that macros can use to determine if a signal ended the job and from which macro the signal number can be derived.	If a program is not enabled to use signals, nothing happens. Otherwise, it depends on the program signal handler and the default action of the signal.

In order to facilitate porting. OS/400 has implemented two additional APIs that can be used directly from the program and offer different kinds of diagnostic details depending on the need. These APIs are QCMDEXC, which basically is only passed the command string and the command length, and QCPACMD, where it is possible to prompt the user for a command, to validate the command, and to invoke it. QCPACMD also has considerable support for returning diagnostic data. Of course OS/400 also supports the wait() WIF... macros, but only as a part of the regular wait() management handling.

It should be remembered that all of these methods only are to be used when a new program is to be called from the current program. No new process is created, no parent-child relationship is valid, and the main program pauses until the called program has returned. Special care has to be taken regarding signal control (blocking masks and signal handlers) when either or both of the two involved programs are using this functionality. More information about how to deal with signals in this kind of environment is found in 4.5.3.3, "Scope of Signal Information" on page 61.

Environment variables are inherited between the program invocations.

If a program was heavily dependent on the UNIX implementation of the system() call, it is very much possible to try to emulate it. Maybe not entirely but to some extent, since it would also mean some kind of emulation of fork(). Here is an example of how it can be done:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>    /* malloc, free */
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>
#include <qp0z1170.h>  /* Environment */

#define ARGNUM 15   1

int mysys(const char *cmdstring);

int mysys(const char *cmdstring)
{
  extern char **environ;

  pid_t pid;
  int   status;
  struct inheritance inherit;
  char *spw_argv[ARGNUM];
  int  numarg = 0;
  char *command;

  if (!cmdstring)    2
    return 1;

  if(!(command = calloc(1, strlen(cmdstring))))
  {
    perror("malloc");   3
    return 1;
  }

  if (QpOzInitEnv() < 0)
  {
    perror("QpOZInitEnv");    4
    return 1;
  }

  strcpy(command, cmdstring);

  for(spw_argv[numarg++] = strtok(command, " ");    5
      ((spw_argv[numarg++] = strtok(NULL, " ")) &&
      (numarg <= ARGNUM)););

  memset(&inherit, '\0', sizeof inherit);

  if ((pid = spawnp(spw_argv[0], 0, NULL, &inherit,
                  spw_argv, environ)) < 0)
  {
    perror("spawn");
    free(command);
    return 127;    6
  }

  while(waitpid(pid, &status, 0) < 0)
    if (errno != EINTR)
```

```
        {
          status = -1;   7
          break;
        }

      free(command);
      return status;   8
    }
```

Figure 18. System() Emulation. This program emulates UNIX system() to a certain extent by starting a new (child) job to run the program.

This function spawns a child process, which is added to the same process group as the parent. Additionally, it is eligible to receive signals. These signals, however, must adhere to the OS/400 signal model, that is, `raise()` does not work.

Additionally, the process environment is inherited by the child since the function resolves the `environ` variable and forwards it. If the program has any open file descriptors, they are also inherited by the child. If that is the case, in order to be able to use them, they are passed as arguments to the program or in the environment.

Notes:

1 Maximum number of arguments that can be sent by the calling program. Any more arguments are discarded.

2 If NULL is passed, 1 is returned to indicate that a shell is available, if we can allow ourselves to consider the OS/400 command interpreter as a shell.

3 Since `strtok()` is rude enough to destroy the received command string, we take the liberty of copying it into a dynamically-created memory area to be `free()`d later.

4 The `environ` external environment pointer is resolved.

5 The arguments passed to the program must be separated into a vector in order for `spawn()` to properly take care of them. `strtok()` is used, because it disregards multiple spaces between arguments, and separates tokens into C strings by ″\0″ terminating them.

6 This is an interesting choice since `spawn()` emulates the combination of `fork()` and `exec()`. Remember that in UNIX, if `fork()`

fails, -1 is returned and if exec() fails, 127 is returned. If somebody disagrees, they are free to adapt the source themselves.

7 If waitpid has failed, return -1, unless it was interrupted (typically by a signal).

8 Return the status received from waitpid(). This value can be used in any of the macros specified in <sys/wait.h>. Of course, it is really of little value to see if the child process was stopped. Since this implementation does not make use of the WUNTRACED option, which is neither implemented in the standard UNIX system() function nor in OS/400, the mysys() call is not returned if the child process was stopped (using SIGSTOP or HLDJOB).

The major difference between mysys() and the system() calls in UNIX and OS/400 is that this function wants the name of an OS/400 *program*, not an OS/400 *command*. If you want to modify it to accept commands instead, you have to use either one of the APIs referred to on page 76, or interestingly enough, the ILE C/400 system() function inside of mysys(). Of course, that is not always the optimal solution, since it causes an extra layer of programs to be added between the current job and the program to run and sometimes this leads to unwanted side effects in terms of return values and diagnostics if one of the middle layer programs fails.

The following example shows how a program can call this slightly altered system():

```
#include <stdio.h>
#include <stdlib.h>
#include <qp0z1170.h>
#include <sys/types.h>
#include <sys/wait.h>

#define system mysys

int mysys(const char *cmdstring);

void main(int argc, char **argv)
{
  char *pathvar = "PATH=%LIBL%";   1
  int  status;

  if (argc != 2)
  {
    fprintf(stderr, "%s: Invalid number of arguments.",
            argv[0]);
    return;
  }

  if (putenv(pathvar) < 0)   2
  {
    perror("Putenv");
    return;
  }

  status = system(argv[1]);

  if (WIFEXITED(status))
    printf("Normal termination: %d.\n", WEXITSTATUS(status));
  else
    if (WIFSIGNALED(status))
      printf("Abnormal termination. Signal %d.\n",
             WTERMSIG(status));
    else
      if (WIFSTOPPED(status))
      printf("Child Stopped. Signal %d.\n",
             WSTOPSIG(status));
  return;
}
```

Figure 19. Program to Use Altered System(). In this sample, WIFSTOPPED() is used, even though it is never called. The reason is, of course, to illustrate how the wait() related macros are used.

The ENV program shown in Figure 17 on page 72 is called by using `CALL <program> PARM(env.pgm)`. The program DSPVAR (Figure 16 on page 71) is called using `CALL <program> PARM('DSPVAR.PGM name')`, where `name` is an environment variable for which a value has been defined.

Notes:

1 We set the PATH to the current value of the library list. Since spawnp() is used, it allows the system to find the program as long as the correct library is added to the library list.

2 The PATH is added to (or replaced in) the environment.

Usually it is quite sufficient to use the system() provided with ILE C/400. Not only is it faster and requires less system resources, but it is provided with the C runtime and does not require additional code to be written. The preceding examples were not only introduced to allow applications dependent on system() return status information a somewhat easier way of porting their OS/400 code, they were also provided as a means for programmers to understand the process flow, inheritance, environment variables, and other components related to how OS/400 handles relations between processes.

4.6.4.2 SBMJOB (Submit Job)

SBMJOB is the traditional OS/400 method to create a batch job. It allows the user to specify what command to run, which job description to use, and a substantial amount of job related attributes. Jobs submitted by SBMJOB are *not* considered to be children of the process from which they were submitted. As a result, it is not possible to wait() for the submitted process or inherit file descriptors. Even though SBMJOB creates a completely new process, environment variables *are* inherited in V3R6 of OS/400. However, that is *not* the case in V3R1.

However, SBMJOB *does* give you the flexibility of being able to change job attributes such as output queues, job queue priorities, and a lot of very useful options. Additionally, it also, by default, submits jobs to the QBATCH subsystem. Using spawn(), the child job is run in the same subsystem as the parent, which causes batch jobs to run in the QINTER subsystem. This might not always be a good idea from a system resource point of view. Even without using SBMJOB, it is possible for a process to change its own attributes by issuing one of the CHGJOB (Change Job), RRTJOB (Reroute Job), or TFRJOB (Transfer Job) commands. Since this book primarily is about application porting, work management in OS/400 is not addressed. A recommended source of information is the *AS/400 Work Management Guide*. SC41-3306.

4.6.4.3 Spawn()

Spawn() is *the* way OS/400 implements a parent-child relationship. It is mentioned in the POSIX.1d draft, previously known as POSIX.4b. The following is a short section of spawn(), which has been extracted from Joe Gwinns (<GWINN@SUD2.ED.RAY.COM>) report on the April 24-28, 1995 meeting in Irvine, Ca.:

″...POSIX.1d, around 130 pages in length, contains a number of realtime interfaces and options that arrived too late to be included in 1003.1b- 1993 (which itself consists of POSIX.1 and POSIX.4 combined). The major new interfaces and options are: spawn(), a functional merger of fork() and exec(), needed both for efficiency and to allow use on platforms lacking memory management hardware;...″

In OS/400 V3R1, spawn() is the *only* way of inheriting descriptors such as environment variables, signal masks, and signal vectors from one process to another. It is also the only way of causing the SIGCHILD signal to be sent to the parent process group when the child process terminates. V3R6 allows, as mentioned in 4.6.4.2, "SBMJOB (Submit Job)" on page 81, environment variables to be inherited but nothing else.

The spawn() family is similar to the fork() and exec() combination. It requires some setup by the programmer in terms of setting up child argument parameters in a vector, somewhat similar to execv.() calls or more specifically, execve() if spawn() is used. The spawn() family also includes spawnp(), which essentially works the same way with the exception that it uses the PATH to find the program it is to start. In the execv.() family, there is really no equivalent of the spawnp() functionality of:

1. Passing the argument format in an array.
2. Passing the environment manually.
3. Searching the PATH.

Regarding some details and examples of how spawnp() can be set up to use the OS/400 library path, see page 80, page 128, and 5.3.5.3, "OS/400 using Spawn()" on page 126.

Spawned child processes are batch jobs. They cannot call interactive commands on the terminal or use system() (see 4.6.4.1, "System()" on page 74 for more information). This is the opposite of what happens if *threads* were used. By using the concept of CPA service threads, multiple jobs within the process are allowed to communicative interactively. The child job has the same user profile, library list, and run attributes as the parent job. This includes the use of the same subsystem, which implies that if the parent is an interactive job in QINTER, the children can, in effect, become batch jobs running in QINTER, which can cause a negative performance impact on a system.

As mentioned earlier, the child process must be passed the environment variables manually, that is, they are not inherited by default. If the child is to inherit all of the parent′s environment variables, the environ external variable can be used as the value for envp[] when spawn() is called. An example of this is found in the program listed in Figure 19 on page 80. If a specific set of environment variables is required in the child, the user must build the array with the "name=value" strings. This causes spawn() to perform the equivalent of putenv() on each element of the envp[] array.

The program that is to run in the child process must be either a program object in the QSYS.LIB file system (a *PGM object), or a shell script.

The only way to execute the equivalent of a UNIX shell script in OS/400 is to use spawn(). It is important to realize that OS/400.. currently does *not* implement shells or shell interpreters. If a shell script is specified to run in the child process, the user must provide such an interpreter as a part of the functionality. The shell script must be a text file and contain a #!<path) <options> in the first line of the file. This allows the program specified as path to be called.

The syntax provided to spawn() must be the proper syntax for the file system in which the program or shell script resides. For example, if the program ENV (Figure 17 on page 72) in library JOHANLIB residing in the QSYS.LIB file system is to be called, the name of the *file* parameter is /QSYS.LIB/JOHANLIB.LIB/ENV.PGM. Note that this specification does not assume the use of the PATH. The result of the call is, therefore, exactly the same, regardless if spawn() or spawnp() is used. As soon as spawnp() notes that there is a "/" (slash) in the filename, it does not consider or evaluate the PATH variable. Instead, it functions identically to spawn() and considers the parameter a valid path to the program or shell script.

As previously mentioned, spawn() allows for the inheritance of file descriptors and socket descriptors. However, it also allows for some flexibility regarding how these are inherited. Basically two methods can be used:

Simple Inheritance This method is closest to the UNIX equivalent of inheriting descriptors. Basically *all* open descriptors are inherited from the parent to the child. This is accomplished by specifying NULL in the *fd_map* parameter. The number of the descriptor in the parent process (for example, number 2) might denote an open file and have the same function and mapping in the child process. Of course, the information has to be passed one way or another. Usually the environment or the child arguments are the most commonly used alternatives.

Mapped Inheritance Mapped Inheritance is a little more complicated. It allows the user to let the child only inherit a subset of the open descriptors, as well as map them explicitly to certain child process descriptor numbers. Mapped inheritance also allows for multiple child descriptors to refer to the same parent descriptor resource.

The way this is performed is (as is the case for arguments and environment) to build a mapping array. The following example is shown:

```
fd_map[0] = SPAWN_FDCLOSED;
fd_map[1] = 0;
fd_map[2] = fd;
fd_map[3] = SPAWN_FDCLOSED;
fd_map[4] = fd;
```

This example causes the child descriptors 0 and 3 to be closed. The parent descriptor 0 is mapped to the child descriptor 1 and the resource specified by the parent descriptor *fd* is accessed using the child descriptors 2 and 4. Remember that if a file descriptor refers to an open instance in a file system that does not support file descriptors in two different processes pointing to the same open instance of a file, the descriptor is closed in the child process.

Additionally, the following applies when spawn() is used:

- Only programs that expect arguments as NULL-terminated strings are spawned.

- The child process is enabled for signals. See 4.5, “Signals” on page 55 for details. A side effect of this function is that the parent process is also enabled for signals if it was not enabled for signals before this function was called.
- If this function is called from a program running in user state and it specifies a system-domain program as the executable program for the child process, the call fails.
- The spawned process is run in the activation group of the program that is the target of the `spawn()`. It may be the same named activation group, but it is not shared in any way with the parent's.
- The new process is added automatically to the parent's process group. See 4.4, “Process Groups and Job Control” on page 46 for more details about process groups.

Just the same as `fork()` in UNIX, the OS/400 implementation of `wait()` and `waitpid()` can be used with `spawn()`. An example of how this can be performed is found in Figure 19 on page 80. It is important to remember that, just the same as in UNIX, it is important to take proper action of what is to happen when the child process terminates. The most common method is to specify SIG_IGN as action for the `SIGCHILD` signal, but the use of `wait()` is appropriate if the parent wants to be asynchronously notified when the child has ended.

4.6.5 Process Authorization

Traditionally, there have been some differences between UNIX and OS/400 in regard to how process authorization, user accounts, group accounts, and adopted authority have been managed. This chapter does not go into detail about file permissions, since it is mostly outside the scope of how process authorization is performed, even though, admittedly, the reason why this concept exists in the first place is to eventually be able to access a resource such as a file and IPC mechanism or some other kind of resource. Additionally in POSIX.1 and most UNIX systems, the authorization of the current process is usually intertwined with the file permission of the executable program.

In a sense, several IDs are associated with a process. These are:

Real uid	
Real gid	Authorization based on what we have logged on as and our primary group. The primary group can usually be determined by entering the command `groups` on the

	command line. The first group displayed is the primary. The remaining groups are supplementary.
Effective uid	
Effective gid	
Supplementary gids	Used for file and IPC access permission.
Saved set-uid	
Saved set-gid	Saved by `exec()` function.

An ID (uid or gid) is basically only a number (in C programs an integer) representing a user or a group of users. Traditionally, the uid 0 (zero) means "root" access or "super user" authority. It is, however, perfectly legal to give any user uid 0 and thereby provide other users with the same authority.

A *user name* is an alphanumeric representation of the uid. When a user logs on, a mapping is performed between the user name and a real uid/primary gid. If multiple entries are found for the user, the most common procedure is that only the first entry is used and the following is disregarded. The *real* uid identifies the user who initiated the current program; usually it represents the uid associated with the login name, but the real uid can be changed during a session using the `setuid()` call.

A group is basically a number of users. Different operating systems have different limits of the number of groups a user can be a member of. There is always one primary group for a user, which usually is referred to as the real *gid* (group ID) when initiating the process. Similar to the real uid, the gid is eligible to change dynamically using the `setgid()` call.

An optional feature of POSIX.1 is the possibility to have *supplementary* group IDs. The value _SC_NGROUP_MAX in `unistd.h` should refer to, if such is the case, the appropriate number of groups allowed per user. If it is not supported, the value of zero (0) should be specified. Before this feature existed, it was necessary to change groups explicitly using different commands such as `newgrp`.

The *effective* uid usually reflects a temporary state, where the functionality of a program demands that a certain uid is performing the operation, usually depending on file permissions if a file has to be updated or an IPC mechanism, such as shared memory, has to be updated by a user, who normally is not authorized to do that certain kind of operation. However, it also has this effect when a file or IPC object is created. It is usually up to the designer/programmer to specify exactly during what parts of the program the effective uid is really needed. Sometimes it can mean some less-wanted side effects to have an effective uid of `root`. In some UNIX operating systems, it is not possible (for example, by default), to let `root` or a superuser on one machine write to an NFS directory mounted on another, while it is perfectly valid for a regular user to perform the same operation.

Consider the example where two or three processes are putting trace messages in a shared memory area. If the shared memory area does not exist, the program creates it. The designer of the program did not want anybody not properly authorized to tamper with the shared memory outside of the program. For this reason, when the shared memory area is created, the owner of the area should be user name *opc*. Additionally, make certain that the permission to the area is 600; that is, the owner can read and write, but nobody else can do anything. However, everyone on the system should have access to the program and use it to update the shared memory.

So how can this be performed...?

A simple way to handle this problem is to:

1. Change the owner of the program to *opc* by issuing the command `chown opc <program>`.
2. Set the *set-user-ID* bit of the program by issuing `chmod u+s <program>`. This makes the permission bits look similar to:

   ```
   $ls -l pgrp
   -rwsr-xr-x   1 opc      tracker      12307 Jun 17 15:57 pgrp
   ```

   The permission gives the user initiating the program the *effective user ID* of the owner of the shared memory area.
3. However, the designer does not want the effective user ID to be that of *opc* during the whole processing. It is enough to change the euid just before accessing the shared memory segment and then reset it back to the real uid.

   The first thing to do in the code is to save the effective uid by calling:

   ```
   int save_uid = geteuid();
   ```
4. When this has been performed, the effective user ID is set back to the real uid. This can be done by using `setuid(getuid())`. When it is time to perform the critical operation, it is a good idea to switch back to the effective uid by issuing `setuid(save_uid)` again.

5. The operation on the shared memory segment is now performed. There is no need to have an effective uid other than the real uid, so the effective uid is once again reset as described in Step 4.

Now, here are some things to bear in mind. The scenario described in Step 4 on page 87 does *only* work on systems with the `_POSIX_SAVED_IDS` symbol set to true. This symbol indicate the existence of the *saved set-user-ID* functionality. It is possible to perceive it as an extra copy of the effective uid, which can be used as previously described by making it possible to change back to the original effective user ID. If this functionality is not available, we have to wait until we can reset the effective uid after the shared memory operation is done. The chain of events then look similar to:

1. Start program. Real uid is the user ID; effective uid is the one for *opc*.
2. Perform shared memory operation.
3. Call `setuid(getuid())`, which resets the effective uid to the real equivalent.

This method also has been the only one applicable if the *opc* user has a uid of 0, which makes him a superuser. The reason is simply, that if a superuser performs the `setuid()` function, not only the effective uid, but also the saved set-uid is changed. The result is that the process does not have the proper permission to change the effective uid back to *opc* when it is time to update the shared memory.

Most or all of the preceding reasoning also applies to gids (group IDs). It is possible to set an effective group ID on a file by issuing the command `chmod g+s <filename>`, and real and effective group IDs are manipulated in the same manner as if they were user IDs. They do, however, differ slightly in how they might be handled, since we are talking about both the primary gid and supplementary gids. When, in the preceding example, you start the program, you automatically get the effective uid of *opc*, but the gid does not change. In order to perform this, either the set-gid bit should be set or the `setgid()` call can be performed. But what happens to the supplementary groups? If the effective uid is changed and the effective gid is changed, will not all the supplementary gids somehow follow in their steps? Unfortunately not. To initialize or set supplementary gids, you either have to use `setgroups()` or `initgroups()`, which in turn uses `setgroups()`. In order to perform this, however, the effective users must be superusers; that is, they have to have uid zero (0).

4.6.5.1 The Way of AS/400 Operating System

OS/400 has usually been considered as a very capable operating system in regard to authorization. However, the price paid is that sometimes it is hard to overlook if a certain approach works in an IS environment. The *OS/400 Security - Reference V3R1* manual, SC41-3301, has been kind enough to supply flowcharts that explain exactly how authorization checking is performed. Unfortunately, the number of flowcharts is currently nine (9), which makes it impossible to provide them in this book as well. We do not go into detail about what levels of security are provided. Instead, we concentrate on what is needed to port applications implementing the UNIX behavior previously described.

Before V3R1 of OS/400, a user profile (an object, that roughly corresponds to a uid (user name set of attributes)) could be a member of only *one* group. However, V3R1 of OS/400 now supports one primary group and 15 supplementary groups. This is verified by examining the NGROUPS_MAX entry in `<limits.h>`, or by calling `sysconf()` with the `_SC_NGROUPS_MAX` parameter. The programmer should be aware that the compile time macro `_POSIX_NGROUPS_MAX` does *not* reflect the actual number of available groups, but is to be considered a minimum.

A group is commonly known as a group profile. A group profile is just a user profile, such as any other user profile and by default, anyone who knows the password is eligible to logon using that account. The recommended approach, however, is to set the password of the group profile to *NONE, which prevents any user from logging on.

Similar to most UNIX systems, V3R1 also provides a primary group concept on an object level. In other words, it is possible to define the group DEPT142 as primary group for a file. Just as it is possible to do `chgrp <newgrp> <file>` in UNIX, it is possible to give the command `CHGOBJPGP` (Change Object Primary Group) on the OS/400 command line.

Assigning a primary group to an object is not mandatory, but can provide a performance advantage compared to having private group authority when the operating system verifies the access level to the object.

Now we are closing in on the interesting things, as seen from a porting point-of-view. OS/400.. does *not* support the set-uid-bit or the set-gid-bit when assigning permissions to an object. The `chmod()` function, which *is* supported, does not accept the `S_ISUID` or `S_ISGID` bits used for implementing the support for manipulating the effective uid and effective gid support. Actually OS/400 does not support the concept of effective uids and gids at all, but from a cosmetic point-of- view which I will come back to later. The setuid(), seteuid(), setgid(), and setegid() do not exist at all.

Does that mean that programs heavily relying on the capability of effective ids are not able to run on OS/400? Fortunately, the answer is no. There are really at least two alternate paths to use. One of them is the concept of *adopted authority*.

When the authority to an object is checked, five main steps (and a lot of intermediate steps) are performed. If any of them proves successful, authority is granted. These steps are:

1. Check object authority.
2. Check user′s authority to object.
3. Check group authority to object.
4. Verify public authority.
5. Verify adopted authority.

We are currently only interested in Step 5. For information about any of the other steps, please refer to the *OS/400 Security - Reference V3R1* manual, SC41-3302.

When a program is created using one of the CRTxxxPGM commands or when changing a program (CHGPGM - Change Program), there is a parameter with the keyword USRPRF. It allows one of two values, either *USER or *OWNER. This parameter determines whether the program uses the authority of the owner of the program in addition to the authority of the user running the program.

We can take a small example: User YESSONG does not have any authority to the file CUSTMAST and no *PUBLIC authority is set. This means, that YESSONG cannot display the contents of the file. Nor can user YESSONG do anything else with it. However, the program CUR001 operates on CUSTMAST. It lets the user read records and update them. If CUR001 has the USRPRF value of *USER, it means that it was running using the authority of the user profile that invoked it. If YESSONG uses this program, it sends an escape message indicating insufficient privileges. But the author of the program has instead let the user profile CUSTMASTER be the owner of the program and set USRPRF to *OWNER. CUSTMASTER is authorized to operate on the file and any user using the CUR001 program has the same authorization as CUSTMASTER as long as the program is running.

Now..., CUR001 is calling CUR002 to retrieve some data from a data area with the same permission as the CUSTMAST file. CUR002 has the USRPRF value of *USER. This means that YESSONG cannot call CUR002 from the operating system command line. However, since CUR002 is called from CUR001, it means that CUR001 is higher up on the *program stack*, and by default, this means that the adopted authority has been inherited by CUR002.

Table 13. Adopted Authority Inheritance

Program Stack before CALL:	**Program Stack after CALL:**
QCMD . . CUR001	QCMD . . CUR001 CUR002

Sometimes this is desirable. Especially if the OPM (Original Program Model, see Chapter 6, “Development Environment on AS/400 System” on page 157 for more details) is used and the design of the program involves a lot of calls to other modules. However, sometimes this inheritance might not be what the designer desires. To change the behavior, it is possible to change the program attribute (CHGPGM) USEADPAUT from *YES to *NO. This can also be specified when the program is created. The object types that can be used for adopted authority are *PGM (Program), *SRVPGM (Service Program), and *SQLPKG (SQL Package).

This approach simulates to some extent, the set-uid-bit behavior of UNIX. It sets what can be considered as the effective uid to the owner of the program, even though OS/400 itself does not support the effective uid concept. However, it does not support the capability of changing the effective uid immediately, as with the setuid() call, with less than changing the owner of the program dynamically while the program runs. This, in turn, causes considerable problems for other invocations, since such a change is system-wide, instead of process-wide. For this purpose, another method can be used. If the user has sufficient authority, it is possible to use the SBMJOB (Submit Job) command and specify an alternative user profile as a part of this procedure. The disadvantage is, of course, that:

- One has to call a new program instead of performing setuid() inline.
- The new job runs in batch.

There is no way of emulating the gid-related euid calls in OS/400.

Table 14 (Page 1 of 4). Summary of What OS/400 Supports in Regard to Authorization Subroutines

UNIX Function	OS/400 Implementation	Comment
getuid()	Supported	Returns the user ID of the calling process.
geteuid()	Supported, but always returns the real user ID, since OS/400 does not support effective user IDs.	Returns the effective user ID or the calling process.
getgid()	Supported	Returns the real group ID (gid) of the calling process.
getegid()	Supported, but always returns the real gid.	Returns the effective group ID of the calling process.
getgrgid()	Supported, but remember to test if the result of the operation is NULL. This is the case if you use getgrgid(getgid()) and the current user profile does not have any associated groups. Additionally, if this routine is used in a threaded program, proper consideration must be taken to the fact that it operates using a static memory area.	Looks up the supplied group ID and returns a pointer to a struct group.
getgrnam()	Supported. However, see getgrgid() for advice if used in a threaded environment.	Looks up the supplied group name and returns a pointer to a struct group.
getgroups()	Supported	Fills an array with the supplementary group IDs of the calling process. If the parameter indicating the size of the array is zero, only the number of elements is returned.
getpwnam()	Supported. The OS/400 implementation is based on POSIX.1 and does not include the pw_comment, pw_quota, and pw_gecos members of the struct passwd structure.	Looks up the supplied user name and returns a pointer to a struct passwd. If this routine is used in a threaded program, remember that it is using a static memory area.

Table 14 (Page 2 of 4). Summary of What OS/400 Supports in Regard to Authorization Subroutines

UNIX Function	OS/400 Implementation	Comment
getpwuid()	Supported. See getpwnam() for additional comments.	Looks up the supplied uid and returns a pointer to a struct passwd.
setpwent()	Not supported	
getpwent()	Not supported	
endpwent()	Not supported	The previous three functions can be used when the programmer has to traverse the entire password file. They can be considered as a wrapper, since you cannot always rely on the fact that the /etc/passwd file is used. The setpwent() is used for rewinding or initializing the information needed, getpwent() is used for receiving each entry into a struct passwd structure, whereas endpwent() is used for the purpose of cleaning up. None of these functions are part of POSIX.1. Nor are they implemented in OS/400.
setgrent()	Not supported	
endgrent()	Not supported	

Table 14 (Page 3 of 4). Summary of What OS/400 Supports in Regard to Authorization Subroutines

UNIX Function	**OS/400 Implementation**	**Comment**
getgrent()	Not supported	The same as getgrgid() and getgrnam(), these functions operate on the struct group structure. They retrieve information from the file traditionally known as /etc/group. They allow a wrapper, which allows the actual implementation of this file to be arbitrary. The functionality of these calls is similar to the ..pwent() calls. None of these functions are part of POSIX.1. Nor are they implemented in OS/400.
setgroups()	Not supported	Sets supplementary gids for the process. Must be called with superuser authority. Not included in POSIX.1.
initgroups()	Not supported	Usually used with setgroups(). Traverses the group file usually using the ..grent() routines mentioned earlier and creates an array with the supplementary gids for a certain user name. This array is used in the call to setgroups(). Not included in POSIX.1.
setuid()	Not supported	If called with non-superuser authority, it changes the effective uid to the real uid or saved-set-uid (provided the _POSIX_SAVED_IDS symbol is defined in <unistd.h>). If called with any other ID, it fails. If called with superuser authority, it changes the real uid, the effective uid, and the saved- set-uid to the new value.

Table 14 (Page 4 of 4). Summary of What OS/400 Supports in Regard to Authorization Subroutines

UNIX Function	OS/400 Implementation	Comment
setgid()	Not supported	If the process has appropriate privileges, the real, effective, and saved set-gid are set to the gid supplied. If the process does not have appropriate privileges, the symbol _POSIX_SAVED_IDS is defined in <unistd.h>, and the gid is equal to the real gid or the saved set-gid, the effective gid is set to the supplied value. The real gid and the saved set-gid are unchanged. If the symbol is not defined, the effective gid can only be changed to the real gid.
setreuid()	Not supported	Swaps the real uid and the effective uid of the process. Not defined in POSIX.1.
setregid()	Not supported	Swaps the real gid and the effective gid of the process. Not defined in POSIX.1.
seteuid()	Not supported	Sets the effective uid.
setegid()	Not supported	Sets the effective gid.
getlogin()	Not supported, but can be rather easily implemented by using the RTVJOBA as shown in Figure 10 on page 50 and retrieve the user profile. This method works even though the USRPRF parameter has a value of *OWNER (adopted authority).	Returns a pointer to the user's login name.

Chapter 5. Networking

The open system concept has traditionally been considered to support considerable distributed network capabilities, both from an architectural point of view as well as from a hardware and available software point of view. However, this was not the case to start with. A number of years passed from the early implementation stages of UNIX in the early 1970s until the first network applications were developed around 1976. At that point in time, UUCP (Unix-to-Unix copy program) was developed and its first major release outside AT&T was with Version 7 UNIX in 1978. The typical transport medium was ordinary telephone lines and its typical use was for distributing software and electronic mail. It also played a very important role when transferring USENET news (news feeds) between different systems.

In the early 1980s, a new family of protocols was specified as the standard for the ARPANET, a network including military, university, and research sites sponsored by the Advanced Research Projects Agency project of the Department of Defence. The accurate name for this family of protocols is the "DARPA Internet protocol suite," but it is commonly referred to as the TCP/IP (Transmission Control Protocol/Internet protocol) protocol suite.

As communication hardware (adapters, connections, routers, hubs, and so on...) became cheaper, the earlier serial connections (often using rather slow links, about 9600 bps), were replaced with faster LAN (Local Area Network) connections. The dominant link protocol used was the CSMA/CD protocol over an Ethernet. Later, the IBM token passing link protocol used over a token-ring physical interface became a major contender of the installed LAN base.

As the popularity of distributed systems increased, people added more and more functionality and even protocols. The most popular protocols as well as other related information are published by the IETF (Internet Engineering Task Force). Some widely used application level protocols are FTP (File Transfer Protocol), TELNET, SMTP (Simple Mail Transfer Protocol), SNMP (Simple Network Management protocol) and many more. Protocols for handling security authentication (Kerberos, IP news feeds, and NNTP (Network News Transfer Protocol)) can also be added to the list.

At about the same point in time to the development of the IP based protocol suite, additional communication protocol suites have been created and implemented. Examples are ISO (International Standards Organization) who created the OSI (Open Systems Interconnection) suite, IBM′s SNA (Systems Network Architecture), NetBIOS (Network Basic Input Output System) and NetBEUI (Extended NetBIOS User Interface), and Novell′s IPX/SPX (Internetwork Packet eXchange/Sequenced Packet eXchange).

The different protocol suites offer different advantages and disadvantages. The TCP/UDP/IP/ICMP stack is very popular and widely spread in the open systems area; SNA is the dominant protocol in IBM based networks. Traditionally, SNA was represented by hierarchical host-based networks based on VTAM and with very limited routing capability. This has, however, been improved by the APPN (Advanced Peer-to-Peer Networking) architecture, which allows for dynamic SNA routing. NetBIOS is the most popular suite in LAN operating systems such as Microsoft LAN Manager or IBM LAN Server, but also in, for example, Banyan Vines. IPX/SPX is the protocol used by Novell in their Novell NetWare environment, which is the most widely spread LAN operating system on the market today.

This variety of protocols with different pros and cons resulted in that users purchased what they needed at different points in time. This resulted in a very heterogeneous environment on most sites. Depending on their advantages and disadvantages, different types of equipment were used in a number of different ways, such as technical calculations and administrative tasks, as well as word processors and host-based transaction based systems.

Very few of these machines could originally interact with each other without a variety of complications. UNIX machines used TCP/IP or maybe UUCP, the AS/400 system was shipped in 1988 with native SNA support. IBM compatible personal computers usually were not shipped with any network support at all. If they were to be connected to a LAN however, IPX/SPX or NetBIOS would be the most probable alternatives. The MacIntosh preferred its own protocol - Appletalk. Soon a new market emerged; suppliers saw it as an opportunity to provide their customers with solutions, basically by providing interoperability between the different platforms. This kind of interoperability was sometimes hard to accomplish. Key issues had to be addressed at very low levels, such as the problem of how to share an adapter resource if the system allows many processes to access different kinds of network adapters concurrently. Other issues that had to be addressed emerged when different suppliers provided different kind of solutions; some might write adapter specific code, some might use packet drivers (a standard for interacting with network adapters), some might use

NDIS (the 3Com/Microsoft LAN Manager Network Driver Interface Specification), and some would use ODI, the Novell specification.

Eventually most of these problems were resolved. The reason was entirely based on hard financial facts. It is not possible to sell software that is only capable of using a very small subset of adapters and it is not good business to create drivers for every possible network card. Manufacturers had to make sure that their software and hardware could interact with what customers had installed; otherwise, their market would have been too limited.

But as soon as the situation had stabilized on the platform the products primarily were aimed at, software manufacturers (who had written software for one operating system and maybe sold a great deal of licences) became aware that it might be possible to sell even more licenses if their software could execute in other environments.

This chapter is to address the topic of how to port IP applications (primarily TCP based) from the UNIX environment to OS/400. Most UNIX client/server applications are heavily depending on TCP/IP, not only by tradition, but also because TCP/IP is included for free in most UNIX operating systems. AS/400 client/server applications have for a long time depended on SNA LU6.2, that is, APPC (ICF) and CPI-C, mainly because it was shipped with the operating system. The TCP/IP programming support of the AS/400 system has been improved dramatically since it was originally released. In the early releases, the calls had to be done using Pascal and the demand for a standard socket interface, callable from C, has been considerable.

5.1 TCP/IP in General

TCP/IP (Transmission Control Protocol/Internet Protocol) is a term which is generally used to refer to a specific set of protocols that allow computers to share resources in a network. The name itself, however, can lead to confusion because even though TCP and IP are two of the protocols in the set (usually referred to as a suite), there are other protocols on the same network layer. This means that the term usually is used both for denoting the combination of the TCP and IP protocol (TCP is used to provide some means of reliability to IP) as well as referring to the whole suite of protocols. This is regrettable, but never the less a reality and since these concepts are generally used, we use the same notation in this chapter.

5.1.1 TCP/IP on the AS/400 System

The original TCP/IP Connectivity Utilities/400 product has seen many improvements over the years since its introduction in V1R2 of OS/400 back in 1989. Historically, it had a very modest beginning, but customer demand and a small group of developers kept making enhancements to the product over the years. But, the original code was a port from VM (Virtual Machine, an operating system implemented on IBM′s 370 and 390 architecture machines), where the implementation was based on Pascal. The effect was that all calls to perform the actual communication had to be done using the Pascal/400 product. Pascal/400 was based on the EPM programming model, see Chapter 6, “Development Environment on AS/400 System” on page 157 for details, which meant that it was possible to use the C/400 product to interact with the communications interfaces by linking with the Pascal code, but the actual communications calls had to be done from Pascal. This made it very hard to port existing IP based applications. They had to have a very clear interface between the communication and the logic, and this was not always the case.

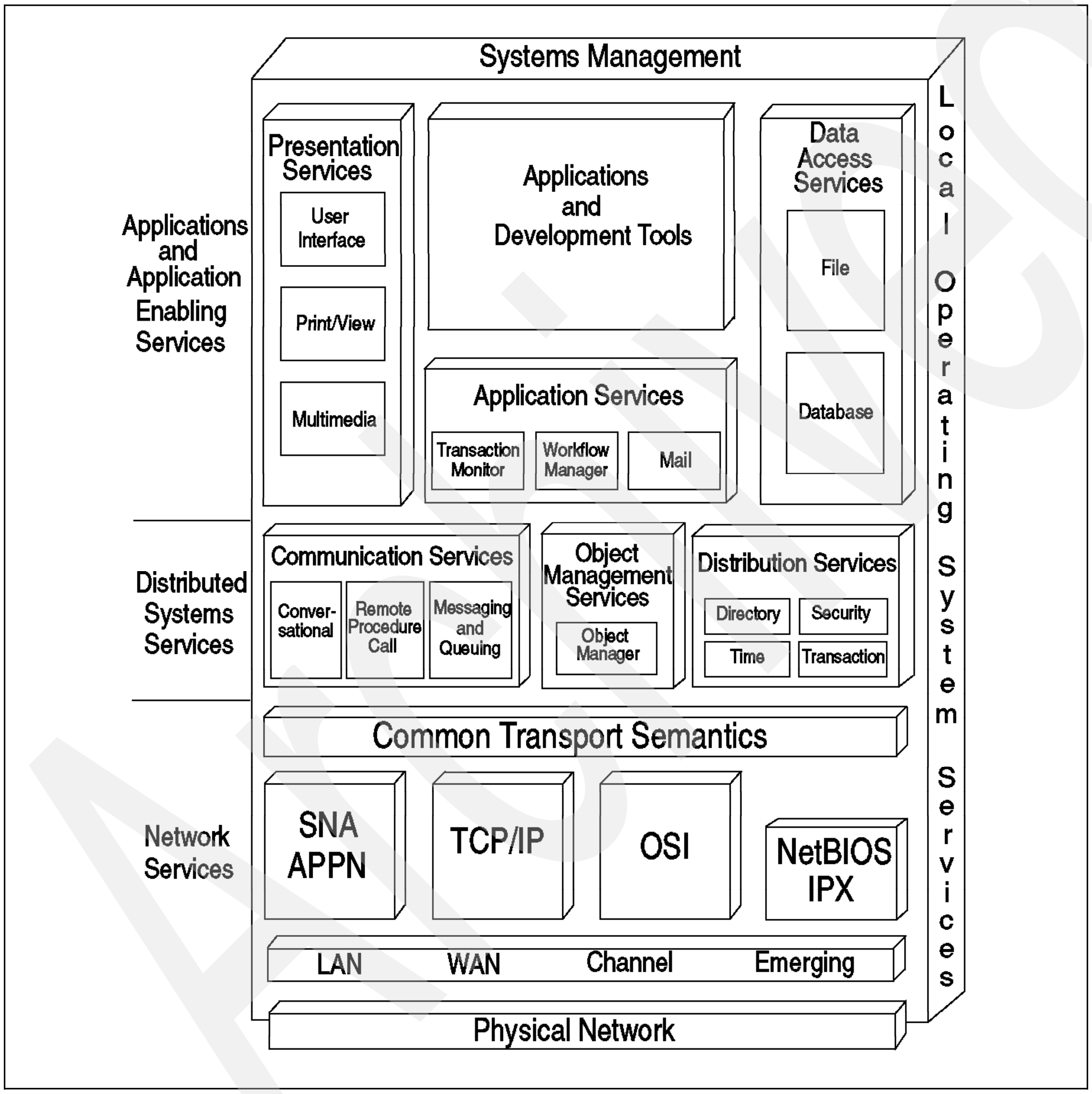
Besides the Pascal API limitation, there were numerous other limitations, such as less than optimal performance, a maximum of 80 IP sessions, a 16MB file transfer limit and the lack of adherence to widely used RFCs (Request for comments). With V3R1 of the operating system, however, the IP functionality has been more tightly integrated with the operating system with additional performance gain as the result. The base has been completely rewritten, the 80 session limit has been removed, and support has been introduced to enable SNMP (Simple Network Management Protocol) Agent and Manager implementations. Numerous other improvements have also made their way into the TCP/IP suite of OS/400, but the most important items from a programmers perspective are:

- The TCP/IP support is now incorporated into the operating system and not a separate product. This means, that software manufacturers no longer have to prerequisite any products for their IP based applications other than the operating system itself.
- OS/400 also provides support for the 4.3BSD socket interface to allow developers to (as painlessly as possible) migrate their current socket-based code to the AS/400 system.

This chapter takes a closer look at methods and techniques from a programmers viewpoint, to port their IP applications from a UNIX platform to OS/400.

5.2 Open Blueprint

The Open Blueprint is the structure that IBM is using to allow the network of systems to function as a unit, as a network operating system. A network operating system is made up of multiple systems that are separated from each other and are connected by a communication network. In the network operating system enabled by the Open Blueprint, each individual logically contains the services described in the following figure. However, it is not necessary for each individual system to physically contain all of the services included in the Open Blueprint. Just as an operating system provides the management of resources on a single system, a network operating system provides for the management across the network of the same type of resources: files, databases, printers, transactions, software packages, documents, jobs and so on. The equivalent facilities or services in each individual system work together to provide support for distributed and client/server applications.



IBM Open Blueprint

The Open Blueprint addresses the challenges of the open environment by viewing a system as part of a distributed network and viewing the network as if it were a single system.

A goal of the Open Blueprint is to provide consistency among IBM products and related products such that they work together to achieve a high level of systemic value. Users usually do not want to be dependent on propriety solutions, since its limits the number of alternatives and choices available for their purposes. This means that there is a huge demand on the market today for openness and product/vendor independence and heterogeneity. The Open Blueprint addresses a combination of existing and emerging industry standards, such as OSF (Open Software Foundation) DCE (Distributed Computing Environment) and Object Management Group Common Object Request Broker Architecture (OMG CORBA).

5.3 Server Models

In many distributed environments, it is common to make use of a Client/Server based way of communicating between different components. Using this model, the task of the server is usually to wait for different kinds of connections, perform some kind of operation, and eventually disconnect. The aim of the client component is usually, at some part during its processing, to connect to the server, ask for or provide certain information, and, at some point in time, to disconnect.

The behavior of a server is also dependent on the transport protocol it uses. The default behavior for LU 6.2 server programs is that they start when an FMH5 (invocation) message is sent. Usually some kind of attach manager runs on the server machine and when a start request arrives, the program is loaded from virtual storage and activated. On the AS/400 system, these kinds of jobs are defined as *prestart jobs*, which starts them in advance and activates them when the FMH5 request arrives. It is similar to the 4.3BSD `rsh`, with the exception that security is really a part of the session layer. The behavior of a protocol suite is thus an important factor when designing how a server program is to behave.

The server program start up behavior is, however, slightly different using IP and, for example, NetBIOS. In both cases, a program has to be running and be prepared for receiving connections. In the NetBIOS case, it issues a *netname*, whereas in the IP case the program waits for a specific *port*. In the TCP case, the server process must accept the client initialization call (`connect()`) before messages can be exchanged between the client process and the server process. In the UDP case, however, the information exchanged is *connectionless*. This essentially means that after the initial socket setup routines, `socket()`, possibly `setsockopt()`, and `bind()`, anything that arrives on that specific port is received. The messages can be sent from any number of client processes, and by using the *From* structure parameter in the recvfrom() call, it is possible to receive information about the message originator.

There are a number of methods for a server process to handle client requests. The kind of implementation chosen can depend on general program design, coding guidelines, how it fits into the operating system, or general skill of the designers and developers. UNIX based operating systems are usually very flexible and allow the designer a number of choices. Some methods have become more popular than others and might even have made it into reusable server libraries for future recommended use.

Of course there exists a number of variations on these schemes, but essentially they are based on a few methods. Some of them are presented in this chapter, both how they look in a UNIX environment, and also some workable approaches of how to port these implementations to OS/400. Before going into detail on each of them, it is however important to understand how OS/400 handles some fundamental concepts that are commonly used in UNIX applications.

5.3.1 Passing of Descriptors

5.3.1.1 UNIX Way

OS/400 handles file and socket descriptors very similar to most UNIX systems. Using the integrated file system functions (see Chapter 3, "File System - AS/400 Integrated File System" on page 25 for details) open() and create(), a file descriptor pointing to the specific file related resource is derived. In a similar manner, the socket() call initializes a socket descriptor for use within an IP application. In UNIX, the most commonly used way of passing open files between processes is to fork() (and possibly exec()) a child process, where all open descriptors are inherited. The child usually closes the open descriptors it does not intend to use and takes advantage of the descriptor or descriptors it does use for further file or socket communication.

OS/400 does not support fork() or exec() but it supports spawn(). It is also possible to use CPA pthreads to emulate a single fork(), (see Chapter 4, "Process Management" on page 41 for details), but it′s better to use the spawn() function to emulate the fork(), exec() combination. The spawn() function tries to emulate the UNIX function as much as possible by passing the open parent descriptors to the child.

Normally, there is no way for one process to pass an open descriptor to some other unrelated process, or for a child to pass an open descriptor to its parent. But 4.3BSD and System V Release 3 both provide a mechanism to do this. The way this is accomplished is presented in detail in other material, such as *UNIX Networking* by Richard Stevens. It is also very closely related to process handling and IPC. Suffice to say, the idea is to pass access right permissions to a certain descriptor using supported IPC mechanisms such as UNIX domain sockets (AF_UNIX).

After this has been done, the `sendmsg()` and `recvmsg()` are used for passing access rights between the processes. OS/400 supports the `sendmsg()` and `recvmsg()` calls, including the *struct msghdr* structure, the MSG_OOB, MSG_PEEK (`recvmsg()`), and MSG_DONTROUTE (`sendmsg()`) flags. However, the passing of access rights is *not* supported. If the *msg_accrightslen* field is not zero and the socket has an address family of AF_UNIX, the function fails with [EOPNOTSUPP]. If the address family is anything but AF_UNIX, that is, AF_INET or AF_NS, the *msg_accrightslen* and *msg_accrights* fields are ignored.

5.3.1.2 AS/400 Way

But does this mean that the AS/400 system is entirely incapable of being able to pass descriptor information between jobs and processes? Fortunately no. There are currently three methods that should do the trick. Not every one of them is always applicable and in the following examples, some general guidelines, indicating when a certain method is preferable, are given. So, in what way can OS/400 assist in passing descriptors?

Method	**Implementation**
CPA threads	This is the closest emulation of socket descriptors from a design point of view if the program does not `fork()` and `exec()` immediately after one another. This option allows some logic to be performed in the child process instead of, or before `exec()`ing a new program. Of course there are some drawbacks. When you are in &unx, you. can call regular C runtime functions or OS subroutines; in the the AS/400 system, you must call a thread function. This can alter the flow of your program, which, most of the time, is not recommended. If you find that this is not desirable, investigate any of the other options. Instead of `fork()`, use `pthread_create()` to start the new thread. However, also be aware that it is advisable to call

pthread_detach to avoid zombie threads. For more information, see Chapter 4, “Process Management” on page 41.

CPA is an optional, but free of charge part of the operating system, which means that no external prerequisites are needed. Take a look at Chapter 4, “Process Management” on page 41 for more details. Socket I/O is *thread-enabled*. This means that the system takes care of resource serialization and access. If thread 1 reads byte number one from a socket and another thread reads from the same socket, the byte accessed is byte number two.

However some functions, such as the functions operating on the *struct servant* and *struct hostent* structures, that is, getservbyname(), gethostbyname(), and gethostbyaddr() use static memory to perform their operations. This means that they are *not* thread-safe. It is up to the programmer to make sure that access to these routines is properly serialized if performed from different threads. Additionally, the socket error number *h_errno* is *not* thread-safe.

By using this method, it is possible to spawn off a new thread each time a connection is established. If the standby job pool is properly configured (see Chapter 4, “Process Management” on page 41 for more details), a substantial performance gain can be achieved using this method compared to spawning a new process. It is important to remember, that in order for a descriptor to be shared between the different threads, it must be static and have global scope.

Also take into consideration if the main thread has any further use of the descriptor passed to the thread or not. If the main thread does an accept() and passes the descriptor to the child, it is very likely that the main thread calls accept() again with the same descriptor to be able to service new clients. An example of this behavior is found in 5.3.5.1, “The UNIX Way” on page 119. Usually the child in these cases wants to dup() the descriptor and not have to depend on the descriptor passed by the main thread. Now the matter of serialization is important. The program must ensure that the child thread has time to perform the `dup()` before the main thread closes the descriptor in order to be able to use the same variable in the next `accept()` call.

This kind of serialization can be accomplished using pthread mutexes, pthread conditions together with mutexes, static variables, and semaphores.

Spawn() `spawn()` is a function that very much resembles the `fork()` function in the sense that descriptors and signal attributes, such as signal mask and signal vector as well as environment variables, are inherited by the child process. The major differences are that:

1. The `fork()`ed child processes are recognized by what the `fork()` routine returns; if it is 0, the process is a child and if it is a positive number, the process is the parent and the number is the process number of the child. The `spawn()` can only return -1 for a failure, or a positive number, which (similar to `fork()`), reflects the child process ID.
2. When using `fork()`, both processes continue processing on the statement directly following the call. Using spawn, the user must supply the name of a program that is to be executed in the child process.
3. The call semantic is different. The `spawn()` demands substantially more parameters in order to be successful.

More information about `fork()`, `spawn()`, and other process related functions is found in Chapter 4, “Process Management” on page 41. The important thing to remember, though, is that an open descriptor is inherited by the child, which means that in existing logic where the parent usually closes its descriptor and reuses it, the child is using the ″still open″ file.

Give/Take descriptor The `givedescriptor()` and `takedescriptor()` calls provide a way for unrelated processes to pass open file descriptors. This might also be used if a child process has created a descriptor it wants to pass to its parent. Its prime target, however, is to compensate for the lack of access right permissions in the `sendmsg()` and `recvmsg()` calls. As mentioned previously, these calls required an AF_UNIX socket (stream or datagram based), where the access information and descriptor address were passed. The `givedescriptor()` and `takedescriptor()` are not that picky. The process that is to provide the recipient with the open descriptor must explicitly specify the internal job identifier in the `givedescriptor()` call. This information has to be passed, but any arbitrary method can be used. Data queues are recommended, but there is nothing that prevents you from using, for example, shared memory, message queues, or other IPC mechanisms.

The internal job identifier can currently *only* be acquired using a work management API, such as QUSRJOBI. It does not have relation to the *process ID* or job number of the job. If the target job in its `takedescriptor()` call specifies NULL as source job identifier, it accepts the descriptor from any arbitrary outstanding `givedescriptor()` request if there are multiple source jobs. If there is no such outstanding request, `takedescriptor()` blocks.

There are, however, some authority concerns that have to be met. The source job giving the descriptor must have the proper authority to give the descriptor to the target job. This allows for impressive flexibility in regard to:

- The current user profile (real uid).
- The group profile, which is not an effective uid, but instead an authority that takes affect if the current user profile does not have sufficient privileges.
- The effective uid, which is represented by the USRPRF keyword when creating or changing a program. This is very much the same as when the set-4 user-ID bit is used in UNIX file permission settings.

Any of these three levels in the source program must match any of the levels in the target program in order for `givedescriptor()` to work.

If both the source job or the target job end before the descriptor has properly been delivered, the system reclaims the resources the descriptor represents.

5.3.2 Standard Descriptors

Usually UNIX developers are accustomed to the fact that three file descriptors are setup for their use in advance. These are STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO (0, 1, and 2). These are by default set to the FILEs *stdin* (by default keyboard), *stdout* (terminal), and *stderr* (terminal). For example, the call `write(STDOUT_FILENO, buf, strlen(buf))` causes the contents of *buf* to be written to the connected terminal. Likewise, the call `fputs(buf, .stdout)` results in the same output.

In OS/400 however, these default descriptors are *not* connected to anything in particular. This can be verified by allocating a descriptor through a call such as `open()`, `creat()`, `pipe()`, `socket()`, or anything similar. It is possible (and even plausible) that the descriptor returned is in the range 0 to 2. Additionally, the *#define*s STD.._FILENO, which usually reside in unistd.h, do not exist in OS/400, which for some programs makes it necessary to create these as a part of the porting effort.

The direct implications are that it substantially limits the porting possibility for applications heavily dependent on this kind of association. For file related programs that might want to `dup2()` input or output to their own routines, more information is found in Chapter 3, “File System - AS/400 Integrated File System” on page 25. From a communication point of view, this has a specific impact when using the inetd daemon. More information about this is found in 5.3.7, “Inetd (The Super Daemon)” on page 134.

5.3.3 Traditional TCP/IP Server Designs

Based on the behavior described in 5.3.1, “Passing of Descriptors” on page 104 and in 5.3.2, “Standard Descriptors,” special considerations must be taken when porting functions that make use of this kind of support. Sometimes it might be very easy just casting a variable, but it can just as well mean *#ifdef*ing large pieces of code out and adding new functionality.

For this reason, a good development strategy is to isolate networking calls as much as possible. The ideal method is to have most communication code in one, or a few modules, since this reduces the number of concerns you must take into consideration. What follows is some of the most commonly used server processing mechanisms in UNIX and how they are implemented in OS/400.

The methods covered are:

Method	**Description**
Spawning a new program	When a client request is received, the program starts a new copy of itself, which performs again all communication related initialization activates and starts listening for new client connections. The parent process continues servicing the request it received. Details are found in program srv1, 5.3.4, "Spawning a New Program" on page 114.
Inherited descriptors	When a client request is received, the program lets its child inherit the open service descriptor and service the client. The parent continues to listen for new requests. Details are implemented in program srv2, which is found in 5.3.5, "Inherited Socket Descriptors" on page 119.
Descriptor arrays	Only one process listens for clients, performs services for existing clients, and cleans up system resources when the connection has ended. Details are implemented in program srv3 in 5.3.6, "Descriptor Arrays" on page 129.
Inetd	The server program makes use of the Inetd daemon and does not implement any socket calls. Details are found in program srv5 in 5.3.7, "Inetd (The Super Daemon)" on page 134.
Passing descriptor access	An open descriptor is passed between two unrelated processes, or between child and parent. No inheritance is involved. Details are found in programs srv4a and srv4b in 5.3.8, "Passing Descriptor Access Permissions" on page 139.

The common behavior of all code examples is that the program is to wait for a client connection on the PORTNUM port. When a connection has been made, they all use different means of handling the current request and still are ready to accept new client connections. The only application logic is to accept an input line and echo it back. If TELNET is used as a client, take care of verifying that it is working in line mode, otherwise each character is echoed back.

Note that the code in the examples is not necessarily to be considered recommended or even good. The only purpose of it is to demonstrate different server program algorithms. The programs were initially compiled and run on an AIX 3.2.5 system.

There is some standard routines that these examples use. The purpose of them is to set up an AF_INET stream socket connection and to do basic read and write as a part of the program logic. These functions look similar to:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>      /* close, read, write    */
#include <sys/socket.h>  /* AF_INET, SOCK_DGRAM...*/
#include <netinet/in.h>  /* sockaddr_in           */
#include <string.h>      /* strstr                */

rcode srv_ini(int *sd, char *buf);
rcode read_des(int des, char *buf);

#define BUFLEN  25
#define PORTNUM 8889
#define HOLDNUM 5
const char      QUIT[] = "\x51\x55\x49\x54";

#define EXIT_RC(a, b, c) {perror(a); close(b); return c;}
  1

typedef enum rc
{
  RC_OK, RC_SOCKET, RC_BIND,
  RC_LISTEN, RC_ACCEPT, RC_CLOSE,    2
  RC_EXEC, RC_READ, RC_WRITE,
  RC_FORK, RC_SEND, RC_RECV, RC_UNIX
} rcode;
```

Figure 20. Standard Header. The information in this header is used by many of the code samples.

Notes:

1 This is a macro used for issuing an error message if it is called after an I/O operation. Additionally, it closes the provided descriptor and returns with the provided reason code.

2 The return codes of the program.

```
/*********************************************************************/
/* Reads information from socket and echos it back.
 * If input has the #define QUIT in it, return.
 */

rcode read_des(int des, char *buf)
{
  int numbyte;

  while(!strstr(buf, QUIT))    1
  {
    if ((numbyte = read(des, buf, BUFLEN)) < 0)
      EXIT_RC("Read", des, RC_READ)

    if (write(des, buf, numbyte) < 0)
      EXIT_RC("Write", des, RC_WRITE)
  }

  close(des);
  return RC_OK;
}

/*********************************************************************/
/* Initializes server socket and clears buffer
 */

rcode srv_ini(int *sd, char *buf)
{
  static struct sockaddr_in server;
  int sock_opt=1;

  if ((*sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    EXIT_RC("Socket", *sd, RC_SOCKET)

  setsockopt(*sd, SOL_SOCKET, SO_REUSEADDR,    2
             (char *)&sock_opt, sizeof(sock_opt));

  memset((void *) &server, '\0', sizeof(struct sockaddr_in));
                                                   3
  memset(buf, '\0', BUFLEN);

  server.sin_family      = AF_INET;
  server.sin_port        = htons(PORTNUM);   4
  server.sin_addr.s_addr = htonl(INADDR_ANY);

  if (bind(*sd, (struct sockaddr *) &server,
           sizeof(struct sockaddr_in)) < 0)
    EXIT_RC("Bind", *sd, RC_BIND)

  if (listen(*sd, HOLDNUM) < 0)
    EXIT_RC("Listen", *sd, RC_LISTEN)

  return RC_OK;
}
```

Figure 21. Read Loop and Stream Socket Server Initialization

These functions, sometimes with slight modifications in the calls, are used by the samples in this chapter.

Notes:

1 This is the main read() and write() logic. It is of little interest, but is mentioned here in order to refer to it later. Notice that in socket programming, all read()s and write()s should be loops, since there is no guaranteed that all of the information is read in one read() or written using one write(). The example does not take this into consideration, since it has little to do with what the example illustrates.

Note, however, that the constant QUIT is represented in hexadecimal form instead of as a character array. The reason is, of course, that the server program is to work on OS/400 as well and compiling the string "QUIT" on the AS/400 system causes the EBCDIC representation to be generated. This means that it is very hard to end the read(), write() loop from an ASCII client (such as TELNET).

2 This is usually used to let the local process portion of a connection association be reused by the bind() call. For example, in the case of FTP, to let the client open a connection server, using the ephemeral port numbers provided by the system for the client control connection. In this particular case, it is used because the TCP behavior usually lets an address exist for twice the maximum lifetime of a packet in the network. In reality, this means that even though a proper close() call is used, it takes one or two minutes before the program can be started anew.

3 Use the SYSV libc function memset(). The ILE C/400 runtime does not use the BSD related bzero(). Note that it is very important to clear the sockaddr_in structure, especially on the AS/400 system. If it is not properly cleared, strange errors might appear such as errno EINVAL. The same rule applies to the BSD function bcopy(). Instead, the ANSI compliant function memcpy() should be used.

4 *Never* forget to filter the integer representation of the port number through htons() in order to convert from host byte order to network byte order. If this is not done, problems can occur when porting to systems where the byte order between local host byte order and standard network order is different.

Fortunately, it does not have a very big impact on the AS/400 system if this is omitted, since the OS/400 byte order implementation does not differ between the two different representations.

5.3.4 Spawning a New Program

A simple way of designing a server process is to let one program wait for a connection call from a client. When the connection has been done, the program spawns a new copy of itself (using fork() and exec()) in order to be able to handle new connection requests while the previous (parent) process continues and handles the communication logic.

This method is very straight forward. No descriptors have to be passed between processes, since every process contains all of the communication logic and is a separate stand-alone program. There is no interaction what so ever between parent and child besides the unwanted side effect of a SIGCHILD in UNIX. This was also the way IP processes (such as FTP) were implemented in OS/400 releases prior to V3R1. When an FTP client was connected, the program quickly submitted a new job that started to wait on the predesignated port, whereas the main process continued to serve the client.

It is usually rather straight forward to port servers implemented this way in OS/400 since very few operating system-specific constructs have to be used. However, most designs are optimized for performance, application flexibility, and maybe legacy code. The process of forking is not very trivial to UNIX based operating systems, but it takes even more resources for OS/400 to create a new job and allocate the associated resources. Additionally, the obvious disadvantage is that in the time span between spawning the new process and until the new process has started to *accept()* new calls, no client can connect.

5.3.4.1 The UNIX Way

```
 .
#include <stddef.h>      1
#include <sys/signal.h>
 .
int main(int argc, char **argv)
{
  int rc;
  int sd, ld;
  char buf[BUFLEN];
  struct sigaction sig_parms;    1

  if ((rc = srv_ini(&ld, buf)) != RC_OK)
    return rc;

  if ((sd = accept(ld, 0, 0)) < 0)
    EXIT_RC("Accept", sd, RC_ACCEPT)

  close(ld);  2

  sig_parms.sa_handler = SIG_IGN;         1
  sigemptyset(&sig_parms.sa_mask);
  sig_parms.sa_flags = 0;
  sigaction(SIGCHLD, &sig_parms, NULL);

  switch(fork())
  {
    case -1:
      EXIT_RC("Fork", sd, RC_FORK)

    case 0:
      close(sd);    3
      execlp(argv[0], argv[0], 0);
      EXIT_RC("Exec", sd, RC_EXEC)

    default:
      if((rc = read_des(sd, buf)) != RC_OK)
        return rc;
  }
  return RC_OK;
}
```

Figure 22. Spawning a New Process on UNIX. The program closes all available descriptors before fork()*ing to prevent them from being inherited by the child. All communication setups must be performed once again for each new job.*

Notes:

> **1** Since the main program spawns a child process and does not use any of the wait() family calls, we must prevent the creation of zombie processes. It should be said that sigaction() is the POSIX way and is likely to be more portable than signal().
>
> **2** This close() call is very important for this server approach. If the listening descriptor is not closed, it is not possible to spawn the program again, since it causes two processes to wait for INADDR_ANY on the same port. When the program is exec()ed, the descriptor opens again.
>
> **3** In the child process, we close the service descriptor, since it is otherwise going to be inherited and this is not what we want for the simple reason that we have no use for it, in addition to giving the system back the resources it represents.

We do not have to do very much in order to make this program work on the the AS/400 system. As specified in 5.3.1, "Passing of Descriptors" on page 104, OS/400 does not support exec() or fork(). This means that these constructs must be *#ifdef*ed out. It is usually very tempting during porting to look for the constructs you want to keep and try to place the *#ifdef*s so the surrounding logic is removed, but the few lines you want to keep are still available.

This reasoning can be applied to the fork() statement in the source. We do not want to keep this piece of logic, but it is very tempting to use the call to read_des(). To do this, however, you also have to comment out the curly bracket connected to the fork() statement, which causes an impact on the logic. It is probably rather harmless in this small example, but remember that if the code is more complex and many curly brackets have to be *#ifdef*ed out, the code eventually is very hard to maintain. Instead, it is recommended to *#ifdef* blocks of code with a matching number of curly brackets.

So, what options are available to port this source to OS/400? There are really two ways this can be achieved: using system() or using spawn(). The differences are:

1. When using system(), there is no need for any signal handler, since the new process does not generate a SIGCHILD when it is terminated. The signal handler is still needed using spawn().
2. Using system(), there is no need to close the service descriptor (derived from accept()) before starting the new OS/400 job, since it is not inherited from the parent process. The descriptor still has to be closed using spawn().

5.3.4.2 The Way of OS/400 Using System()

```
 .
 .
#ifndef __ILEC400__
#include <stddef.h>
#include <sys/signal.h>    1
#else
#include <stdlib.h>      /* system()              */
#define EXECLEN 100
#endif
 .
 .
#ifndef __ILEC400__
  struct sigaction sig_parms;
#else                    2
  char executable[EXECLEN];
#endif
 .
 .
#ifndef __ILEC400__
  sig_parms.sa_handler = SIG_IGN;     3
  sigemptyset(&sig_parms.sa_mask);
      EXIT_RC("Exec", ld, RC_EXEC)
 .
 .
    default:
      if((rc = read_des(sd, buf)) != RC_OK)
        return rc;
  }
#else
  sprintf(executable, "SBMJOB CMD(CALL PGM(%s)) JOB(%s)",
      argv[0], strchr(argv[0], '/') + sizeof(char));
  system(executable);     4

  if((rc = read_des(sd, buf)) != RC_OK)
      return rc;
#endif

  return RC_OK;
}
```

Figure 23. Spawning a Process on OS/400 Using System(). An example of what is needed to port the creation of a new job (UNIX process) to OS/400 without descriptor inheritance.

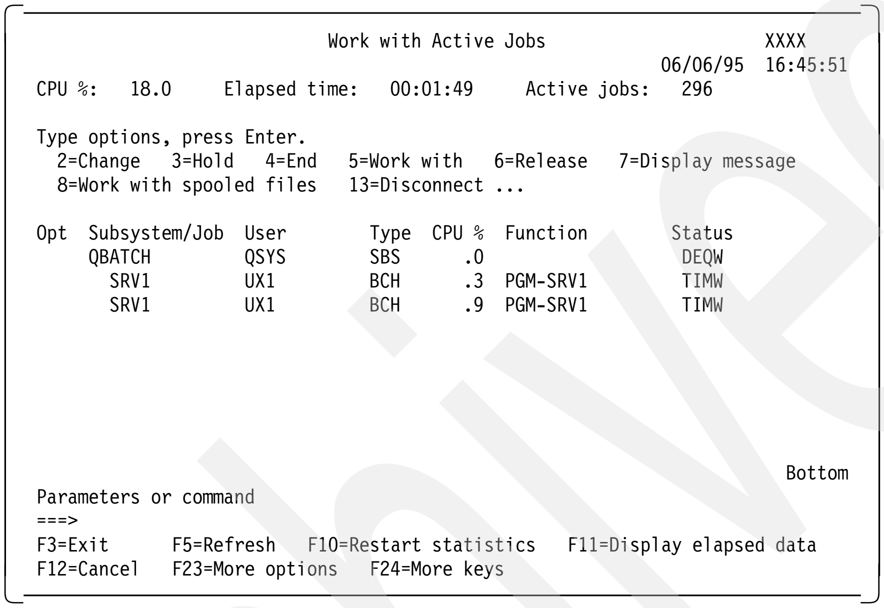
Notes:

1 We are removing the include files needed for signal handling and adding the ones used for `system()`. Since the SBMJOB (Submit Job) command does not cause any child processes (in the UNIX sense) to be generated, such as the `spawn()` function, there is no need to capture SIGCHILD, since it is never sent. Additionally, a constant used for specifying the length of the command used to create a new job is added.

2 The struct sigaction structure is removed and space is allocated to store the command to be invoked.

3 We remove the whole `fork()` construct together with the signal support.

4 Since we removed 3, it is necessary to start the new job some other way. Here we use the `system()` call, but the QCMDEXC API could also have been used. The SBMJOB command is placed in the character array, which is passed to the `system()` call. When this has been done, the parent process continues to service the previously established connection.



```
                             Work with Active Jobs                    XXXX
                                                             06/06/95  16:45:51
CPU %:   18.0     Elapsed time:   00:01:49     Active jobs:   296

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...

Opt  Subsystem/Job  User        Type  CPU %  Function        Status
     QBATCH         QSYS        SBS      .0                   DEQW
       SRV1         UX1         BCH      .3  PGM-SRV1         TIMW
       SRV1         UX1         BCH      .9  PGM-SRV1         TIMW






                                                                      Bottom
Parameters or command
===>
F3=Exit      F5=Refresh   F10=Restart statistics   F11=Display elapsed data
F12=Cancel   F23=More options   F24=More keys
```

Figure 24. WRKACTJOB SBS(QBATCH). This is how the srv1 program looks as one job is connected to an AIX client. At the time of connection, it spawned a copy of itself, ready to serve the next client.

5.3.5 Inherited Socket Descriptors

One of the most commonly used methods for a server to receive client connections in UNIX is for the server to maintain its listening descriptor and when an incoming request is received, spawn a child process and let it inherit the service descriptor (derived from accept()). The parent process continues by closing its copy of the service descriptor and resumes its waiting for new connections. The structure of the program is similar to the one in Figure 22 on page 116 with the following main difference:

5.3.5.1 The UNIX Way

```
 .
 .  /* This code is following the call to srv_init() in
 .    * srv1.c */
 .
while (1)
{
  if ((sd = accept(ld, 0, 0)) < 0)
    EXIT_RC("Accept", ld, RC_ACCEPT)    1

  switch(fork())
  {
      case -1:
        EXIT_RC("Fork", sd, RC_FORK)

      case 0:    /* Child Process  */
        close(ld);    2
        if ((rc = read_des(sd, buf)) != RC_OK)
          exit(rc);
        else
          exit(RC_OK);

      default:   /* Parent process */
        close(sd);   3

    }   /* switch  */
  }     /* while   */

  return RC_OK;   4
}
```

Figure 25. Inherited Descriptors in UNIX. This example represents the traditional approach where the new child inherits the service descriptor. The parent closes it after creating the child and continues to listen to new requests.

After the connection has been established (1), the process fork()s a copy of itself, thereby letting the child process inherit the open file descriptor. The main process could very well close its file descriptor (3) and start accepting new connections, whereas the child process continues to talk to the client (2).

In addition to Figure 25, sometimes, instead of just fork()ing, the program starts a child program, similar to the one in Figure 22 on page 116. The main difference is that, in this case, we want the service descriptor to be inherited by the child process to let it continue the conversation with the client program. In UNIX, the changes in the server program are similar to:

```
 .
{
  .
  char fd[2];
  .
    switch(fork())
    {
      case -1:
        EXIT_RC("Fork", ld, RC_FORK)

      case 0:
        close(ld);
        sprintf(fd, "%d", sd);  1
        execlp("srv2c", "srv2c", fd, 0);
        exit(RC_EXEC);

      default:
        close(sd);

    }   /* switch  */
  .
```

*Figure 26. Inherited Descriptors and Exec(). Instead of handling the conversation in the current program, we start a new program (*srv2c*), which inherits the* sd *descriptor. Note that the* ld *descriptor is closed, which means that it is not inherited.*

As shown in **1** we call the child program with the descriptor number supplied on the command line. It is also possible to use the putenv() function to set an environment variable, which also is inherited. The child program picks up the descriptor as argv[1], and continues the processing:

```
 .
static rcode read_des(int des);
 .
int main(int argc, char **argv)
{
  if (argc != 2)
  {
    printf("%s: Invalid number of arguments. Exiting.\n",
           argv[0]);
    return 1;
  }
  else
    return read_des(atoi(argv[1]));
}

static rcode read_des(int des)
{
  int numbyte;
  char buf[BUFLEN];

  while(!strstr(buf, QUIT))
  {
    if ((numbyte = read(des, buf, BUFLEN)) < 0)
      EXIT_RC("Read", des, RC_READ)

    if (write(des, buf, numbyte) < 0)
      EXIT_RC("Write", des, RC_WRITE)
  }
  close(des);
  return RC_OK;
}
```

Figure 27. Program Inheriting Open Descriptor. This program is called by the program in Figure 26 on page 121. It receives the descriptor on the argument line and uses it for reading and writing.

As described in 5.3.1, "Passing of Descriptors" on page 104, neither of these attempts work in OS/400. In this case, there are other alternatives to choose from:

To illustrate how this is accomplished, an example is presented:

5.3.5.2 OS/400 using Threads

The threaded method is usually preferable from a performance point of view. Since all threads (including the main or primary thread) share the same global static storage and heap, there is no need to *copy* the environment in the way fork() does it since the data is available anyway. This implies some other considerations as well... When the program in Figure 26 on page 121 is closing the service descriptor (sd) in the main process after the child has been created, the operating system only closes *one* instance of the socket resource. The child program had a copy of the same resource and it is not affected by the close().

In Figure 28 on page 125, the thread does not access a copy of the resource; it actually accesses the resource itself. If the main thread closed the service descriptor after creating the thread, the descriptor is also closed for the thread. This is also the case if the thread dup()ed the descriptor, since it does not really copy the resource, but merely provides a pointer.

Additionally, one of the major advantages of using threads, in comparison to other methods of creating concurrently running units of execution in OS/400, is that all of the code can be in one executable module instead of having to create multiple programs (*PGM).

```
#ifdef __ILEC400__
#include <pthread.h>    1
void *process(char *buf);
#endif

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>      /* close, read, write    */
```

```
#include <sys/socket.h>  /* AF_INET, SOCK_DGRAM.. */
#include <netinet/in.h>  /* sockaddr_in          */
#include <string.h>      /* strstr()             */
#include <stdlib.h>      /* exit()               */

#include <stddef.h>
#include <sys/signal.h>

static int read_des(int des, char *buf);

#define BUFLEN  25
#define PORTNUM 8889
#define HOLDNUM 5
const char    QUIT[] =
"\x51\x55\x49\x54";

#define EXIT_RC(a, b, c) {perror(a); close(b); return c;}

typedef enum rc
{
  RC_OK, RC_SOCKET, RC_BIND,
  RC_LISTEN, RC_ACCEPT, RC_CLOSE,
  RC_EXEC, RC_READ, RC_WRITE,
  RC_FORK
} rcode;

int main(int argc, char **argv)
{
  int rc;
  int sd, ld;
  struct sockaddr_in server;
#ifdef __ILEC400__
  char *buf;
  pthread_t thrno;
#else
  char buf[BUFLEN];
  struct sigaction sig_parms;
#endif
  int sock_opt=1;

  if ((ld = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    EXIT_RC("Socket", ld, RC_SOCKET)

  setsockopt(ld, SOL_SOCKET, SO_REUSEADDR,
             (char *)&sock_opt, sizeof(sock_opt));

  memset((void *)&server, '\0', sizeof(struct sockaddr_in));

  server.sin_family    = AF_INET;
  server.sin_port      = htons(PORTNUM);
  server.sin_addr.s_addr = htonl(INADDR_ANY);

  if (bind(ld, (struct sockaddr *) &server,
           sizeof(server)) < 0)
    EXIT_RC("Bind", ld, RC_BIND)

  if (listen(ld, HOLDNUM) < 0)
    EXIT_RC("Listen", ld, RC_LISTEN)
```

```
#ifndef __ILEC400__
  memset(buf, '\0', sizeof(buf));
  sig_parms.sa_handler = SIG_IGN;
  sigemptyset(&sig_parms.sa_mask);
  sig_parms.sa_flags = 0;
  sigaction(SIGCHLD, &sig_parms, NULL);
#endif

  while (1)
  {
    if ((sd = accept(ld, 0, 0)) < 0)    2
      EXIT_RC("Accept", sd, RC_ACCEPT)

#ifndef __ILEC400__
    switch(fork())
    {
      case -1:
        EXIT_RC("Fork", ld, RC_FORK)

      case 0:
        close(ld);
        if ((rc = read_des(sd, buf)) != RC_OK)
          exit(rc);
        else
          exit(RC_OK);

      default:
        close(sd);

    }   /* switch  */
#else
    buf = calloc(1, BUFLEN);    3
    memcpy(buf, &sd, sizeof(sd));
    pthread_create(&thrno, pthread_attr_default,
                   (pthread_startroutine_t) process,
                    buf);
    pthread_detach(&thrno);   4
#endif
  }     /* while   */

  return RC_OK;
}

/***********************************************************************/
#ifdef __ILEC400__
void *process(char *buf)
{
  int myfd;

  memcpy(&myfd, buf, sizeof(myfd));

  if (read_des(myfd, buf) != RC_OK)
    printf("read_des failed.\n");

  free(buf);
  pthread_exit((void*) NULL);    5
  return;
}
#endif
```

Figure 28. Descriptor Inheritance in OS/400 using Threads. The threads support allows the program to respond faster to new connections as well as having the ability to use only one program instead of two, as when using spawn() which uses at least two.

Notes:

1 All programs using the CPA pthreads must include `<pthreads.h>` first in the source file, since it includes wrapper functionality needed to enable existing functions to perform in a threaded environment.

2 The program blocks on the `accept()` call. When a connection is established, it wakes up. Note that all UNIX related `fork()` logic has been removed.

3 Remember now, that we can have multiple connections. If each one of them uses the same area of memory to read from and write to, this leads to unpredictable results. What we are doing instead, is allocating a buffer for each thread. The `calloc()` is used instead of `malloc()`, since it clears the allocated space for us (fills it with '\0's).

The alternative is to use two allocate `bufs` in the thread (`process()`). However, this adaptation of the program uses `buf` to pass the socket descriptor to the thread. So, why do we have to pass the descriptor..? Why can't we just use a global descriptor and copy it to the threads automatic storage? This is basically a way out of solving a resource problem. If `sd` was a global variable and `process()` copied it to automatic storage as soon as the thread was spawned and active, we have a time slot between the moment of `pthread_create()` and the moment of copying the variable.

During this time slot, a new client might connect, which means that `sd` was overwritten before it was copied.

4 The `pthread_detach()` causes the threads control blocks and resources to be deleted after the thread ends. If this is not done, and a `pthread_join()` is not performed, zombie threads come in to existance.

5 This call makes sure that the system considers the current thread as a candidate for the standby pool, which causes subsequent thread creations by the process to be performed significantly faster. See 4.6.3, “Threads” on page 73 for more details about threads, zombie threads, and the standby pool.

5.3.5.3 OS/400 using Spawn()

Whereas 5.3.5.2, “OS/400 using Threads” on page 122 more closely resembles the pure fork() based logic in Figure 25 on page 120, the following example is build upon the OS/400 equivalence of exec(), (Figure 26 on page 121). It is built upon the use of the spawn(), which initializes a new program with options to inherit environment, signal masks as well as descriptors. More information is found in Chapter 4, “Process Management” on page 41. The spawn() can supply descriptor inheritance in two ways:

Simple inheritance All open descriptors are automatically inherited. Simple inheritance is the method that most resembles its UNIX equivalence and we are going to use it for the following sample (Figure 29 on page 127). By using simple inheritance, the descriptors receive the same descriptor number for a descriptor resource (file or socket) in the child process as they had in the parent process.

Mapped inheritance The programmer explicitly specifies which descriptors are to be inherited. It is also possible to specify how the descriptor numbers in the parent should map against the numbers in the child.

```
 .
#ifdef __ILEC400__
#include <qp0z1170.h>   1
#endif

#ifdef __ILEC400__
#include <spawn.h>
#define  CHILD "YSRV2C.PGM"   2
#define  ARGNUM 3
#define  ENVNUM 1
#endif

int main(int argc, char **argv)
{
 .
```

```
  char fd[4];
#ifdef __ILEC400__
  struct inheritance inherit;
  char *spw_argv[ARGNUM];   3
  char *spw_envp[ENVNUM];
  char *pathvar = "PATH=%LIBL%";
  memset(&inherit, '\0', sizeof inherit);
#endif
 .
    if ((ld = accept(sd, 0, 0)) < 0)
      EXIT_RC("Accept", sd, RC_ACCEPT)

#ifdef __ILEC400__
    if (putenv(pathvar) < 0)   4
    {
      close(ld);
      EXIT_RC("putenv", sd, RC_EXEC)
    }
    sprintf(fd, "%d", ld);
    spw_argv[0] = CHILD;
    spw_argv[1] = fd;   5
    spw_argv[2] = (char *) NULL;
    inherit.pgroup = 0;
    spw_envp[0] = (char *) NULL;
    if ((rc = spawnp(CHILD, 0, NULL, &inherit,
           spw_argv, spw_envp)) < 0)
    {
      printf("rc is %d.\n", rc);
      close(ld);
      EXIT_RC("spawn", sd, RC_FORK)
    }
#else
    switch(fork())
    {
 .
    }   /* switch  */
#endif
    close(ld);
  }     /* while   */

  return RC_OK;
}
```

Figure 29. Descriptor Inheritance in OS/400 using Spawn(). The traditional fork(), exec() flow has been altered to allow for what is needed for spawn().

As is shown, the entire `fork()` clause has been *#ifdef*ed out. But the sources have many similarities, for example, the procedure of passing information about the open descriptor to the child through the command line. The immediate advantage is that the source in Figure 27 on page 122 does not have to be changed at all between the UNIX environment and OS/400. Since it is a very simple program and does not do anything except read from and write to a descriptor, it is possible to port without performing any changes whatsoever.

Notes:

1 In V3R1 of OS/400, the function prototype to `putenv()` is not to be found in stdlib.h, as usually is the case. No plans for providing this support in a PTF exist at the time of writing this book. See 4.6, "Starting and Stopping Processes/Threads" on page 65 for details about environment variable handling in OS/400.

2 The name of the program to start. Note that even though the name of the program is `YSRV2C`, a `.PGM` has to be appended.

3 Initialization of variables used for inheritance, command line parameters, and the environment. Additionally `pathvar` is set to `%LIBL%`. This is an interesting piece of mixing the two worlds. UNIX machines by tradition are using a PATH. The AS/400 system is using a *library list*. The library list is a little bit more functional in the sense that it is used not only to find executables, but also to find database files and other operating system objects. Since `spawnp()` is using the PATH to find the executable and we know that it is found using the library list, we set the PATH environment variable to the value of the library list.

This is done at **4**. Note, that it is perfectly valid to remove both of these lines and instead use the operating system command ADDENVVAR (Add Environment Variable) before running the program.

5 We initialize `argv[]` and `envp[]` arguments before supplying them to the `spawnp()` call. The `spawnp()` was chosen instead of `spawn()` since it is functionally more similar to `execlp()` used in Figure 26 on page 121.

5.3.6 Descriptor Arrays

There are other ways to design server applications than just the regular fork(), exec() behavior described in the previous examples. Just imagine an application that does not invoke any child process, but still is able to maintain a connection for multiple clients. There are obvious disadvantages, such as “Is the program going to be able to perform some logic while it is maintaining the communication?.” The answer is, of course, “it depends.” It is not responsible to let this kind of program handle hundreds of requests and still be able to answer all of these requests with good response times on other hardware than very high scale processors. However, with a reasonable (and preferably configurable) number of supported connections, the server program can do more than just listen to the port, perform an action, and return to its silent waiting. Chapter 4, “Process Management” on page 41 specifies how the alarm() call can generate a SIGALRM signal, which interrupts a select() wait and perhaps lets the program perform some internal maintenance jobs. Additionally, the select() call itself allows a maximum wait time to be passed in the struct timeval parameter passed to it.

This approach makes the following program structure possible:

```
                          Start
                            |
                     Initialize program
                            |
   Wait for new clients, existing clients requests or timeout
    |                       |                          |
    |                       |                          |
  < New client >         Accept question        Do service work
    |                       |                          |
  Verify authority       Find out answer       Return to select()
    |                       |
  Add to current socket set  Respond
    |                       |
  Reply affirmative      Return to select()
    |
 Return to select()
```

Figure 30. Descriptor Array Server Logic. Of course, it is possible in the middle path to receive information about disconnected clients. If this is the case, the number of bytes read is usually zero. The “Respond” action is then to remove the disconnected descriptor from the maintained array.

The absolutely major advantage of this approach is that it does not involve any process related functionality at all. Only *one* process is involved. This, in turn, means that it is much easier to port this kind of program to OS/400 as we discover in 5.3.6.2, “Descriptor Array for OS/400” on page 133.

5.3.6.1 Descriptor Array for UNIX

```
 .
#include <sys/select.h>      1
 .
#define MAXCONN 5            2
 .
int main(int argc, char **argv)
{
  int rc, numbyte;
  int sd, ld;
  char buf[BUFLEN];
  int conn[MAXCONN];       3
  int cnum = 0;
  fd_set source, ready;
  int i, sel_event, fdp1;

  if ((rc = srv_ini(&ld, buf)) != RC_OK)   4
    return rc;

  FD_ZERO(&source);    5
  FD_SET(ld, &source);

  while(1)
  {
    6
    fdp1 = high_val(cnum, ld, conn) + 1; /* Find the highest fd */

    /* Copy the contents of the master (source) fd_set
     * to the working (ready) fd_set.
     */
    memset((char *)&ready, (int) NULL, sizeof(ready));
    memcpy((char *)&ready, (char *)&source, sizeof(ready));

    /* Wait for message */
    sel_event = select(fdp1,            /* Size of bit array */
                 &ready,            /* Sockets to listen to */
                 (void*) 0,             /* No Sockets writing */
                 (void*) 0,             /* No Error handling  */
                  NULL);     7      /* Time out period    */

    /*
     * sel_event == 0 for time out,  -1 for interrupt or Error
     * ready now has the socket(s) that have recv’d data
     */
    if (sel_event < 0)
    {
      perror("Interrupt");
      continue;
    }

    /* If any new process has connected, register it. */
    if (FD_ISSET(ld, &ready))
    {
      if (cnum == sizeof conn)
      {
        printf("Maximum number of connections reached.\n");
        continue;
      }

      if ((sd = accept(ld, 0, 0)) < 0)
        EXIT_RC("Accept", sd, RC_ACCEPT)

      printf("New connection - number %d - has been
              established.\n", cnum + 1);
      FD_SET(sd, &source);
      conn[cnum++] = sd;
    }

    for (i=0; i < cnum ; i++)     8
      if (FD_ISSET(conn[i], &ready))
      {
        if (((numbyte = read(conn[i], buf,      9
              sizeof(buf))) <=0) || (strstr(buf, QUIT)))
        {
          printf("Connection has ended for client number %d.\n",
           i + 1);

          /* break the communication: Error or shutdown */
          FD_CLR(conn[i], &source);   10
          close(conn[i]);

          /* move the last socket to the current place */
          conn[i] = conn[--cnum];
        }
        else
          if (write(conn[i], buf, numbyte) < 0)
            perror("Could not write data.");
      } /* if      */
  }     /* while   */

  return RC_OK;
}
```

Figure 31. Descriptor Array on UNIX. This is an example of a server that does not fork but, instead, maintains the descriptors in an array. This approach also makes it possible to store information about the clients if the array consists of structures containing relevant information for the application instead of descriptor integers.

Notes:

1 The sys/select.h is used by AIX, Solaris and Sun-OS, whereas OS/400 and HP-UX have the FD_ macros and the function prototype specified in time.h.

2 Maximum number of clients to accept.

3 Array of socket descriptors.

4 Initiate communication.

5 The *source* is the main fd_set. When a descriptor is added or removed from the array, it is removed in *source*. Before the `select()` call, the contents is copied to the *ready* fd_set. The reason is, of course, that the result of `select()` sets and removes the bits corresponding to each of the available descriptors. This leaves the fd_set unusable if `select()` is called with it the next time. However, by using a master fd_set and copying the contents each time, we can be certain that *ready* is properly initialized.

6 The `select()` function wants to be informed about which number is the highest descriptor it should take into consideration. It is, of course, possible to just enter FD_SETSIZE, which specifies the maximum number of descriptors, but for performance reasons, we lend the kernel a helping hand.

The high_val calculates the maximum descriptor value by traversing the descriptor array and takes the listening descriptor into consideration as well.

7 No timeout. The `select()` waits forever.

8 If a message arrives from any of the connected clients, it must be handled.

9 If the `read()` fails or the client is terminated, we remove it from the socket descriptor array and clear the master fd_set (**10**).

In 6, we explained how the highest used socket descriptor is calculated. The source from high_val is shown in the following figure.

```
/********************************************************************/
/* Calculate the highest used file descriptor in order
 * to use the correct mask in the select statement.
 */

static int high_val(int cnum, int sd, int *conn)
{
  int hval, i;

  if (!cnum)
    return sd;
  else
  {
    hval = sd;

    for (i=0; i < cnum; i++)
      if (conn[i] > hval)
        hval = conn[i];
  }

  return hval;
}
```

Figure 32. High_val.c. This program aids in determining the highest used descriptor number. This value incremented by one specifies the range of descriptors the `select()` *call is watching.*

5.3.6.2 Descriptor Array for OS/400

As mentioned earlier, this approach does not involve UNIX specific process handling routines such as fork() and exec(). The following example specifies the changes in the code that needed to be implemented in order to port from AIX to the OS/400 environment.

```
 .
#ifndef __ILEC400__
#include <sys/select.h>
#else
#include <time.h>
#endif
 .
 .
```

Figure 33. Srv3.c for OS/400. The changes needed to make the example shown in 5.3.6.1, "Descriptor Array for UNIX" on page 130 work properly under OS/400. The differences in where `select()` *is defined is described in* **1** *in 5.3.6.1, "Descriptor Array for UNIX" on page 130.*

5.3.7 Inetd (The Super Daemon)

As we have seen by now, there are a lot repetitive steps that an IP server has to perform each time it must establish a connection. These steps involve creating sockets, binding sockets, listening on a descriptor, accepting the client call, and usually also to `fork()` (and sometimes to `exec()`) when the client request enters the system. Additionally, proper error handling routines must be written to check for failures for each step and take some proper action, which could be closing sockets, sending a message to the SYSLOG, and so on.

However, in most UNIX systems, there is a simpler way to manage this kind of server processing, since they usually provide an Internet (meaning the AF_INET family) superserver (inetd). This daemon mainly provides two features:

- It allows one process to listen on multiple ports, thereby making it unnecessary for programs using this functionality to do all the dirty work themselves. It also spares the system (usually quite considerably) since there is only one process running instead of multiple servers, each waiting on a port of their own.
- It takes care of the usual daemon initialization routines, which includes disconnecting from the terminal and forking an additional copy of itself if this has not been done already.

The price you have to pay is that for each client request, inetd will not only `fork()` but also `exec()` the actual user supplied server program. We do not describe in detail how to configure inetd, since it is very well documented elsewhere. Suffice to say, the inetd daemon is configured to start the user service program when a client request is issued to a particular port. The inetd, which is waiting in a `select()` statement, accepts the call and `fork()`s itself. While the parent program resumes the `select()` call, the child closes all open descriptors except the one used for the current socket. This descriptor is `dup2()`d to descriptors STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO (see 5.3.2, “Standard Descriptors” on page 109) before it is closed.

After this has been performed, the server program specified for the particular service is executed. This means that for the UNIX environment, everything read from stdin or written to stdout/stderr as well as everything read from descriptor STDIN_FILENO (0), written to STDOUT_FILENO (1), or STDERR_FILENO (2) is read or written to the actual original socket. All the user program has to do is to include the usual include files (<stdio.h>, ...) and treat the incoming and outgoing requests as if they were coming from or directed to the terminal.

A typical program, performing the same function as srv1.c and srv2.c, is similar to:

5.3.7.1 Inetd for UNIX

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>      /* close, read, write... */
#include <string.h>      /* strstr, memset...     */

#define BUFLEN  25
#define EXIT_RC(a, c) {perror(a); return c;}

const char    QUIT·“ = ”\x51\x55\x49\x54”;

typedef enum rc
{
  RC_OK, RC_READ, RC_WRITE
} rcode;

int main(int argc, char **argv)
{
  int numbyte;
  char buf[BUFLEN];

  memset(buf, ’\0’, sizeof(buf));

  while(!strstr(buf, QUIT))
  {
    if ((numbyte = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
      EXIT_RC(”Read”, RC_READ)
    buf[numbyte] = ’\0’;

    if (write(STDOUT_FILENO, buf, strlen(buf)) < 0)
      EXIT_RC(”Write”, RC_WRITE)
  }

  return RC_OK;
}
```

Figure 34. Inetd for UNIX. Note that no socket calls whatsoever are used and that the I/O is performed against stdin and stdout.

So, how is it possible to port this kind of program to OS/400? One problem is that inetd is not a part of the operating system. However, this does not mean it is impossible to perform the same functionality. An example of how this can be done is displayed later in this chapter. A more serious problem is described in 5.3.2, “Standard Descriptors” on page 109. If you emulate the inetd functionality using the same procedure as inetd itself, that is, wait on a port, and dup2() the socket descriptor to the 0, 1 and 2 descriptors, it is relatively easy. However, *only programs written directly to these descriptors* work properly. Programs using any of the standard ILE C runtime stdio functionality such as gets(), puts(), printf(), and scanf() do *not* work.

The following example does not emulate the daemon functionality of inetd. All it does is to provide some sample means of being able to use code previously written against any of the standard UNIX descriptors. The deal is that the server program itself calls a function to initialize the 0, 1, and 2 descriptors at the time of initialization. It also cleans up after itself by calling the function with another parameter. Using this method, it is not very hard to write a real inetd daemon with the same functionality.

5.3.7.2 Inetd for OS/400

Include files and #*define*s of BUFLEN, PORTNUM, and HOLDNUM, as well as the definition of the return codes (*rcode*) are found in Figure 20 on page 111.

```
 .  /* Include files and return codes */
typedef enum fcn
{
  FCN_INIT, FCN_CLOSE
} fcn;

rcode ovrfd(fcn func);
#ifdef MAIN
static rcode read_des(void);
#endif

/* &numsigndefine of BUFLEN, PORTNUM and HOLDNUM */

const char  QUIT[] =
            ”\x51\x55\x49\x54”;
#define STDIN_FILENO    0
#define STDOUT_FILENO   1
#define STDERR_FILENO   2

#define EXIT_RC(a, b, c) {perror(a); close(b); return c;}

rcode ovrfd(fcn func)
{
  int ld, loop;
  int sock_opt=1;
  static int sd;
  static struct sockaddr_in server;

  if (func == FCN_CLOSE)
  {
    for (loop = STDIN_FILENO; loop <= STDERR_FILENO; loop++)
      close(loop);
    close(sd);
    return RC_OK;
  }
  if ((ld = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    EXIT_RC("Socket", ld, RC_SOCKET)
  .
```

setsockopt(), memset() server.sin_ assignments, bind() and listen() as shown in the srv_ini function of Figure 21 on page 113.

```
  .
  if ((sd = accept(ld, 0, 0)) < 0)
    EXIT_RC("Accept", ld, RC_ACCEPT)

  close(ld);

  for (loop = STDIN_FILENO; loop <= STDERR_FILENO; loop++)
    if (sd != loop)
      if (dup2(sd, loop) < 0)
      {
        perror("dup2()");
        EXIT_RC("dup", sd, RC_ACCEPT)
      }

  return RC_OK;
}

#ifdef MAIN
int main(int argc, char **argv)
{
  rcode rc;

  ovrfd(FCN_INIT);
  rc = read_des();
  ovrfd(FCN_CLOSE);

  return rc;
}
```

Figure 35. Inetd for OS/400. A simple example of how the inetd behavior, to some extent, can be emulated under OS/400.

The read_des() code is slightly changed from the one in Figure 21 on page 113 in the sense that nothing is passed to it. It works exactly as if it was used in a non-socket environment.

```
static rcode read_des(void)
{
  int numbyte;
  char buf[BUFLEN];

  while(!strstr(buf, QUIT))
  {
    if ((numbyte = read(0, buf, BUFLEN)) < 0)
      EXIT_RC("Read", 0, RC_READ)

    if (write(1, buf, numbyte) < 0)
      EXIT_RC("Write", 1, RC_WRITE)
    if (write(2, buf, numbyte) < 0)
      EXIT_RC("Write", 2, RC_WRITE)
  }

  return RC_OK;
}
#endif
```

Figure 36. Read_des.c. Rread_des() *reads from STDIN_FILENO and writes to STDOUT_FILENO and STDERR_FILENO. The explicit use of 0, 1, and 2 used in the* read() *and* write() *are only there to demonstrate that these are the descriptor numbers actually used.*

5.3.8 Passing Descriptor Access Permissions

In 5.3.1.1, "UNIX Way" on page 104, we mentioned the possibility of passing open descriptors between potentially unrelated processes. It allows for one process to be waiting for socket connections from a client and when a connection has been performed, pass the open descriptor to another process that receives client messages, carries out commands, and sends the response backs. This implies that the first process is a *front end* process since it only takes care of the communication connections. This approach also saves system resources compared to, for example, 5.3.5.1, "The UNIX Way" on page 119, since no process has to fork. It also differs from 5.3.6.1, "Descriptor Array for UNIX" on page 130 in the sense that the process that receives the descriptor does not have to perform any socket initialization whatsoever.

Basically, there are two ways of performing this function. The SVR4 way is to create a stream pipe() and pass the descriptor information using the I_SENDFD and I_RECVFD commands to the ioctl() functions. In 4.3BSD, this information is forwarded using an AF_UNIX connection based or datagram socket and the sendmsg() and recvmsg() functions.

OS/400 does not support either the STREAMS interface or the passing of open file descriptors through sendmsg() or recvmsg(). The two latter functions are included in the operating system, but their only purpose is to pass data between processes.

This means, essentially, that if a UNIX program uses sendmsg() and recvmsg() for passing data, it still works unmodified. However, if they are used for passing access rights to descriptors, other alternatives must be considered. These alternatives are either:

1. Redesigning the program in such a way that this kind of procedure is not necessary.
2. Use the OS/400 functions givedescriptor() and takedescriptor(). An example is shown in 5.3.8.3, " Passing Descriptor Access on OS/400" on page 146.

We assume that the program consists of two processes. Both of them act like daemons in the sense that neither has a user interface, both are disconnected from the terminal, and both are IP servers.

Program srv4a is waiting for connections from a client. When a connection has been established, process srv4b is notified. Program srv4a passes the access permissions to program srv4b, which handles the rest of the communication session and program srv4a resumes its waiting on the port.

Program A is similar to:

5.3.8.1 Passing Descriptors Access on UNIX

```
 . /* Heading from Figure 20 on page 111. */
#include <sys/un.h>      /* UNIX domain           */
#include <sys/uio.h>     /* IO Services           */
 .
#include <sys/select.h>
 .
#define NOFD    -1    1
 .
#define UNDOMNM "/tmp/pipename"    2
#define EXIT_RC(a, b, c) {return cleanup(a, b, c);}

typedef struct fds
{
  int sd;
  int ld;
  int ud;
} fds;
```

```
static rcode cleanup(fds *desc, char *msg, rcode ret);    3

int main(int argc, char **argv)
{
  char proto = '\0';
  fds desc;
  rcode rc;
  fd_set source;
  int fdp1, sel_event;
  struct iovec iov[1];
  struct msghdr msg;
  struct sockaddr_un client;

  desc.ud = desc.sd = desc.ld = NOFD;    1

  memset((void *)&client, '\0', sizeof(struct sockaddr_un));

  client.sun_family = AF_UNIX;
  strcpy(client.sun_path, UNDOMNM);      4
  client.sun_path[strlen(UNDOMNM)] = '\0';

  if ((desc.ud = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
    return cleanup(&desc, "UNIX Socket", RC_SOCKET);

  if (connect(desc.ud, (struct sockaddr *) &client,    5
              (int) SUN_LEN(&client)) < 0)
    return cleanup(&desc, "UNIX Connect", RC_CONNECT);

  if ((rc = srv_ini(&desc)) != RC_OK)
    return rc;

  FD_ZERO(&source);
  FD_SET(desc.ld, &source);

  while(1)
  {
    fdp1 = desc.ld + 1; /* Find the highest fd */

    /* Wait for message */
    sel_event = select(fdp1,            /* Size of bit array */
                       &source,         /* Sockets to listen to */
                       (void*) 0,       /* No Sockets writing */
                       (void*) 0,       /* No Error handling  */
                        NULL);          /* Time out period    */

    /*
     * sel_event == 0 for time out,  -1 for interrupt or Error
     * ready now has the socket(s) that have recv'd data
     */

    if (sel_event < 0)
    {
      perror("Interrupt");
      continue;
    }

    /* If any new process has connected, register it. */
    if (FD_ISSET(desc.ld, &source))
    {
      if ((desc.sd = accept(desc.ld, 0, 0)) < 0)
        return cleanup(&desc, "Accept", RC_ACCEPT);

      printf("New connection has been established.\n");
      iov[0].iov_base = &proto;     6
      iov[0].iov_len  = sizeof proto;
      msg.msg_iov     = iov;
      msg.msg_iovlen  = 1;
      msg.msg_name    = (caddr_t) 0;
      msg.msg_accrights    = (caddr_t) &(desc.sd);
      msg.msg_accrightslen = sizeof desc.sd;

      if(sendmsg(desc.ud, &msg, 0) < 0)    7
        return cleanup(&desc, "Sendmsg", RC_SEND);

      close(desc.sd);
    }
  }

  return RC_OK;
}
```

Figure 37. Passing Descriptor Access on UNIX. This sample connects to a AF_UNIX named socket, receives requests from AF_INET clients, and passes the open descriptor derived from the client connection to the server (Figure 38 on page 146).

Notes:

1 Constant used for un-initialized descriptors.

2 Since the processes are unrelated, the AF_UNIX connection must be named. The means for descriptor passing is the file `/tmp/pipename`.

3 This routine closes all open descriptors. It has nothing to do with the logic of the program and is not listed.

4 Initialization of the sockaddr_un structure used to connect to the AF_UNIX socket.

5 Connect to the AF_UNIX socket.

6 The only thing we want to do now is to send over the descriptor. This is easily arranged by not sending any data whatsoever and only initialize the access fields. However, on the recipient side, the `recvmsg()` receives zero bytes. That means, it could mean complications, since some programs usually look to see if the return from `recv...()` functions is zero or negative. In this case, zero is perfectly normal. However, if this program is terminated one way or another, a zero is also received on the recipient side.

Additionally, most programs implement some kind of protocol to specify exactly what is sent, in this case, we specify that if '0' is sent in the data field, access rights are being transferred.

7 The protocol information and access rights are sent.

Before this program is submitted, the AF_UNIX socket server program *srv4b* must be started.

5.3.8.2 Passing Descriptor Access on UNIX Part II

```
 .
#include <errno.h>
 .
 .
static int high_val(int cnum, int sd, int *conn);
static int cleanup(int a, int b, char *msg, int rc);
extern int errno;
 .
#define NOFD    -1
 .
#define QUIT    "QUIT"
 .
 .
int main(int argc, char **argv)
{
  char proto = '\0';
  int numbyte;
  int ud, ld, sd;
  char buf[BUFLEN];
  int conn[MAXCONN];
  int cnum = 0;
  fd_set source, ready;
  int i, sel_event, fdp1;
  struct iovec iov[1];
  struct msghdr msg;
  struct sockaddr_un server;

  unlink(UNDOMNM);
```

```
memset((void *)&server, '\0', sizeof(struct sockaddr_un));

if ((ud = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
  return cleanup(NOFD, NOFD, "UNIX Socket", RC_SOCKET);

server.sun_family = AF_UNIX;
strcpy(server.sun_path, UNDOMNM);
server.sun_path[strlen(UNDOMNM)] = '\0';

if (bind(ud, (struct sockaddr *) &server,
         (int) SUN_LEN(&server)) < 0)
  return cleanup(ud, NOFD, "UNIX Bind", RC_BIND);

if (listen(ud, HOLDNUM) < 0)
  return cleanup(ud, NOFD, "UNIX Listen", RC_LISTEN);

if ((ld = accept(ud, 0, 0)) < 0)
  return cleanup(ud, NOFD, "UNIX accept", RC_ACCEPT);

printf("Connection from UNIX socket client performed.\n");
iov[0].iov_base = &proto;
iov[0].iov_len  = sizeof(proto);
msg.msg_iov     = iov;
msg.msg_iovlen  = 1;
msg.msg_name    = (caddr_t) 0;
msg.msg_accrights    = (caddr_t) &sd;
msg.msg_accrightslen = sizeof sd;


FD_ZERO(&source);
FD_SET(ld, &source);

while(1)
{
  fdp1 = high_val(cnum, ld, conn) + 1;  /* Find the highest fd */

  /* Copy the contents of the master (source) fd_set
   * to the working (ready) fd_set.
   */
  memset((char *)&ready, (int) NULL, sizeof(ready));
  memcpy((char *)&ready, (char *)&source, sizeof(ready));

  /* Wait for message */
  sel_event = select(fdp1,            /* Size of bit array */
                     &ready,          /* Sockets to listen to */
                     (void*) 0,       /* No Sockets writing */
                     (void*) 0,       /* No Error handling  */
                      NULL);          /* Time out period    */

  /*
```

```
 * sel_event == 0 for time out,  -1 for interrupt or Error
 * ready now has the socket(s) that have recv′d data
 */

if (sel_event < 0)
{
  perror(″Interrupt″);
  continue;
}

/* If any new process has connected, register it. */
if (FD_ISSET(ld, &ready))
{
  if (cnum == sizeof conn)
  {
    printf(″Maximum number of connections reached.\n″);
    continue;
  }
1
  if((recvmsg(ld, &msg, 0) <= 0) || (errno == EBADF))
  {
    i = cleanup(ud, sd, ″Recvmsg″, RC_RECV);
    close(ld);
    return i;
  }

  printf(″New connection - number %d - has been established.\n″,
         cnum + 1);
  FD_SET(sd, &source);
  conn[cnum++] = sd;
}

for (i=0; i < cnum ; i++)
  if (FD_ISSET(conn[i], &ready))
  {
    if (((numbyte = read(conn[i], buf, sizeof(buf))) <= 0)
||
        (strstr(buf, QUIT)))
    {
      printf(″Connection has ended for client number %d.\n″,
             i + 1);

      /* break the communication: Error or shutdown */
      FD_CLR(conn[i], &source);
      close(conn[i]);

      /* move the last socket to the current place */
      conn[i] = conn[--cnum];
    }
    else
    {
          buf[numbyte] = '\0';
          if (write(conn[i], buf, strlen(buf)) < 0)
            perror("Could not write data.");
        }
      } /* if     */
  }     /* while   */

  return RC_OK;
}
```

Figure 38. Passing Descriptor Access on UNIX Part II. This program acts as a AF_UNIX socket server and receives the open file descriptor from the client (Figure 37 on page 142). The layout of the program is very similar to Figure 31 on page 132. For additional details about the logic, see 5.3.6, "Descriptor Arrays" on page 129.

Notes:

1 The file descriptor is received.

5.3.8.3 Passing Descriptor Access on OS/400

Now we have an interesting challenge ahead of us. The scheme used in the UNIX implementation has to be altered entirely, and yet we must try not to impact the logic of the program more than absolutely necessary.

One of the biggest obstacles is that the server program (srv4b) is waiting on a `select()` clause. If `sendmsg()` is used, the `select()` reacts properly and returns control to the program, but `givedescriptor()` does not affect `select()` at all. So how can this be solved? Basically, it does not need to be very complicated. It is true, that we have replaced the access right passing, but we still have the protocol character to send. In this program, it is absolutely not necessary for the logic, since the identifier is never used. However, as mentioned in **6** of Figure 37 on page 142, it meets a need from a reliability point of view. In the port to OS/400, this protocol character plays another important role. The `sendmsg()` that transmits the character is not removed from the application logic, instead it triggers the `select()` of the server program and notifies it when it is free to call `takedescriptor()`. If the client had not issued `givedescriptor()`, the `takedescriptor()` call blocks and prevents the server from listening to any of the existing clients.

```
 .
#ifndef __ILEC400__
#include <sys/select.h>
#else
#include <sys/time.h>
#endif
 .
#define EXIT_RC(a, b, c) {return cleanup(a, b, c);}

#ifdef __ILEC400__
#define JOBID_LEN 16
#endif
 .
  struct sockaddr_un client;
  int  datalen;
#ifdef __ILEC400__
  char job_ident[JOBID_LEN];    1
#endif
 .
    return cleanup(&desc, "UNIX Connect", RC_CONNECT);

#ifdef __ILEC400__    2
  if ((rc = read(desc.ud, job_ident, sizeof job_ident))
                  < sizeof job_ident)
    return cleanup(&desc, "UNIX Read", RC_READ);
  datalen = 0;    3
#else
  datalen = sizeof desc.sd;
#endif
  .
      msg.msg_name    = (caddr_t) 0;
      msg.msg_accrights    = (caddr_t) &(desc.sd);
      msg.msg_accrightslen = datalen;

#ifdef __ILEC400__
      if(givedescriptor(desc.sd, job_ident) < 0)    4
        return cleanup(&desc, "Givedescriptor", RC_SEND);
#endif
      if(sendmsg(desc.ud, &msg, 0) < 0)    5
        return cleanup(&desc, "Sendmsg", RC_SEND);
 .
```

Figure 39. Passing Descriptor Access on OS/400. The figure represents changes that were done in order to port the mechanism of distributing descriptors between unrelated processes. A single period (.) on a line means that the code is identical to Figure 37 on page 142. The changed lines are usually preceded with one or two lines from the original UNIX code to put it into perspective.

Notes:

1 This is the internal job ID, which is transferred from the program, that is to receive the descriptor. Note that we here have added an extra piece of information to be sent between the processes. The process, which opens the descriptor and uses givedescriptor() to pass it, must be aware of this internal identifier, which makes it necessary to pass it over. Since we already have means for accomplishing this (the AF_UNIX socket), this demands little more than a regular send/receive operation.

2 The internal job identifier is received.

3 We no longer pass any access rights using sendmsg(). However, this call is still performed to transfer the protocol information as explained initially in this chapter. Since OS/400 implementation generates a -1 return code if the msg_accrightslen member is something else except zero, it is necessary to provide a variable, which has different values depending on the environment.

4 The givedescriptor() call is non-blocking. After it has been performed, the *sendmsg()* wakes the server process from its select() and makes it accept the new descriptor.

5.3.8.4 Descriptor Access on OS/400 Part II

The program that supplies internal job identifiers and uses the handed descriptors is similar to:

```
 .
#include <errno.h>

#ifndef __ILEC400__
#include <sys/select.h>
#else
#include <sys/time.h>
#include <qusrjobi.h>    1
#endif
 .
#define UNDOMNM "/tmp/pipename"
const char    QUIT[] =     2
"\x51\x55\x49\x54";
 .

  struct sockaddr_un server;
#ifdef __ILEC400__
  struct { char jobname[10];
           char usrname[10];      3
           char jobnum[6];
         } qjobname;

  Qwc_JOBI0100_t  jobinfo;    4
#endif
 .
 .
  printf("Connection from UNIX socket client performed.\n");
#ifdef __ILEC400__
  memset(&qjobname, ' ', sizeof qjobname);
  qjobname.jobname[0] = '*';                5
  QUSRJOBI(&jobinfo, sizeof(jobinfo), ″JOBI0100″,
           &qjobname, &(qjobname.usrname));

  if((i = write(ld, jobinfo.Int_Job_ID,      6
               sizeof jobinfo.Int_Job_ID)) <
     sizeof(jobinfo.Int_Job_ID))
    return cleanup(ud, ld, ″Cc_xwrite″, RC_WRITE);

  i = 0;
#else
  i = sizeof sd;    7
#endif
 .
  msg.msg_name    = (caddr_t) 0;
  msg.msg_accrights    = (caddr_t) &sd;
  msg.msg_accrightslen = i;
 .
      if((recvmsg(ld, &msg, 0) <= 0) || (errno == EBADF))
      {
        i = cleanup(ud, sd, ″Recvmsg″, RC_RECV);
        close(ld);    8
        return i;
      }

#ifdef __ILEC400__
      if ((sd = takedescriptor((char *) NULL)) < 0)
      {    9
        i = cleanup(ud, sd, ″Takedescriptor″, RC_RECV);
        close(ld);
        return i;
      }
#endif
      printf(″New connection - number %d - has been established.\n″,
              cnum + 1);
 .
```

Figure 40. Passing Descriptor Access on OS/400 part II

Notes:

1 Since the QUSRJOBI Application Program Interface (API) is used to derive the internal job identifier, a function prototype must be included.

2 And of course QUIT entered from an ASCII terminal does not have the slightest resemblance to the EBCDIC equivalent.

3 The input parameters to the QUSRJOBI API must be instanciated....

4 ... as well as the structure to hold the result.

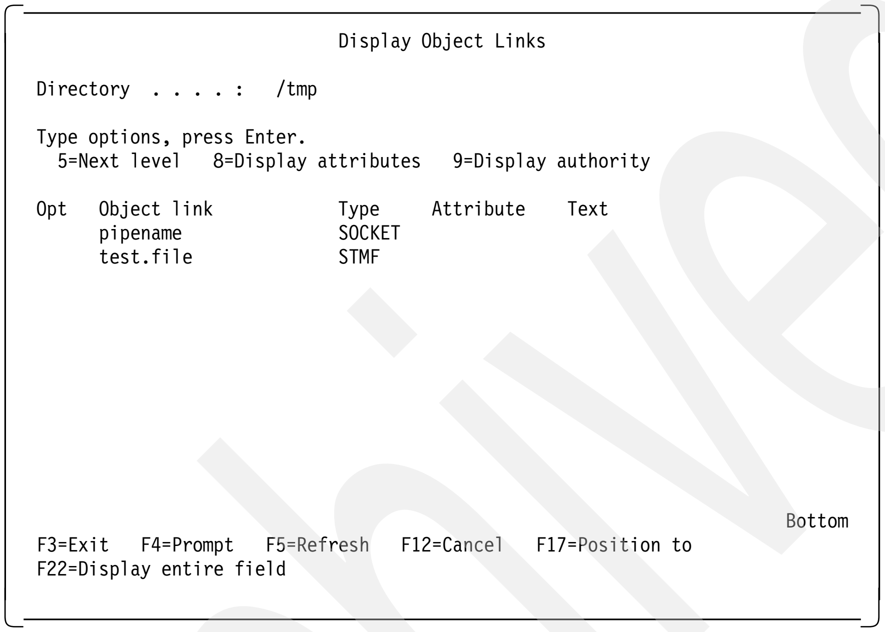
5 Now we have to initialize the API input parameters. By specifying an asterisk (*) in the *Job name* field, we specify that we want information about the current job. Setting this value also means that the rest of the members in the structure have to be blank. The last argument to the API is really supposed to be the *Internal job identifier* if it going to identify the job, but since we use *Job name* for this purpose, it too must be blank. Instead of declaring a separate variable for this, a character array of the same length and contents is used.

6 The derived job identifier is now sent over the process that is going to send us the open descriptor.

7 See **1** in Figure 39 on page 147 for details.

8 The protocol token has now been received; it means that if the `recvmsg()` call succeeds, there should now be a descriptor available.

9 The descriptor is received and a new client session has been established.



```
                              Display Object Links

 Directory  . . . . :   /tmp

 Type options, press Enter.
   5=Next level   8=Display attributes   9=Display authority

 Opt   Object link           Type     Attribute    Text
       pipename              SOCKET
       test.file             STMF











                                                                         Bottom
 F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F17=Position to
 F22=Display entire field
```

Figure 41. Unix Domain File Entry. The preceding `pipename` *is created by the program for UNIX domain communication. Notice that even though it is listed in the directory, it is not a file (STMF). If* `stat` *or* `fstat` *is used, the result shows an file system entry of the type S_IFSOCK.*

5.4 General Tips When Porting Network Applications to OS/400

This chapter is a short summary of what to think about when porting network (IP) applications from UNIX to the AS/400 system. Some items are not limited to just networking related topics, but can have an impact on all types of applications.

Item	**What to Think About**
STREAMS	OS/400 does not support the STREAMS interface. In other words, there is no support for the following `ioctl()` commands: • I_FLUSH • I_SETSIG • I_GETSIG

- I_LOOK
- I_FIND
- I_PEEK
- I_SRDOPT
- I_GRDOPT
- I_NREAD
- I_STR
- I_SENDFD
- I_RECVFD
- I_LINK
- I_UNLINK

Descriptor Access OS/400 does not support the passing of open descriptors using sendmsg() and recvmsg(). The functions are there, but can only be used to pass data. Additionally, as pointed out in and in , the I_SENDFD and I_RECVFD functions cannot be used to pass open file descriptor information as introduced in SVR4.

However, other methods might be used for this kind of open descriptor passing between unrelated processes. See 5.3.1, “Passing of Descriptors” on page 104 for details.

TLI OS/400 does not support the TLI (Transport Level Interface) programming model for network transport layer independence primarily used for the TCP/IP and UDP/IP protocol suite and for the OSI equivalent.

Header File Location This is usually a problem also when porting between UNIX environments. An example is the <sys/errno.h> file, which does not exist on OS/400. It can only be found as <errno.h>. Additionally, for example, <signal.h> must be specified as <sys/signal.h>. Most UNIX systems have a symbolic link from /usr/include to /usr/include/sys for some header files.

RPC	OS/400 supports neither Sun′s Remote Procedure Calls nor Apollo′s NCS (Network Computing System). However, DCE/400 allows for DCE (Distributed Computing Environment) RPC functionality on the AS/400 system.
Contents of header files	In AIX, Solaris and SunOS for example, the <sys/select.h> is needed for the `select()` function prototype and the FD_ macros. On OS/400 and HP-UX, it is enough to include <sys/types.h> and <sys/time.h>.
Variable initialization	Particularly important for structures such as sockaddr_in and sockaddr_un. They should be cleared (`memset()`) before they are used in any subroutine or API call.
Source file record length	The default record length using the CRTSRCPF (Create Source Physical File) on the AS/400 system is 80 bytes + 12 bytes for date and sequence information. It is generally a good idea to use at least 132 bytes (+ 12) to make certain that the file is able to contain the source lines without truncating them.
Compiler defines	The ILE C/400 compiler automatically defines the __ILEC400__ symbol, which can be used in #ifdef constructs.
sys_errlist[]	No sys_errlist exists on OS/400. The information can be retrieved by writing a function, though. If this method is used, sys_errlist must be defined as a macro.
`fork()`, `exec()`	There is no `fork()` and `exec()` in OS/400. Methods of circumventing this problem are found in 5.3.1, "Passing of Descriptors" on page 104.
STDXXX_FILENO	There is no automatic connection between file descriptors 0, 1, and 2 and the FILE structures stdin, stdout, and stderr. Depending on how the application is using this relationship, different methods of circumventing the problem can be used. See 5.3.7, "Inetd (The Super Daemon)" on page 134 for an example.

NLS	OS/400 does not support the XPG3 `catopen()`, `catclose()`, and `catgets()` calls. They can be created using regular operating system message files, though.
`setuid()`	There are some authority process related functions that are not implemented in OS/400 Examples are `setuid()`, `setgid()`, the set-uid bit on files, and `initgroups()` as well as certain pid related functions (see Chapter 4, "Process Management" on page 41 for details).
`bzero()`, `bcopy()`, `bcmp()`	Use the SYSV libc function `memset()`. The ILE C/400 runtime does not use the BSD related `bzero()`. Note that it is very important to clear the sockaddr_in structure, especially on the AS/400 system. If it is not properly cleared, strange errors might appear of which errno EINVAL is only one. The same rule applies to the BSD functions `bcopy()` and `bcmp`. Instead, the ANSI compliant functions `memcpy()` and `memcmp()` should be used.
`htons()`	Do not forget to filter the integer representation of the port number through `htons()` in order to convert from host byte order to network byte order. If this is not done, problems can occur when porting to systems where the byte order between local host byte order and standard network order is different. This is, however, not the case in OS/400.
sockaddr_in, sockaddr_un	On some UNIX operating systems, the `sockaddr_un` and `sockaddr_in` structures have an extra length field. For example, compare `sockaddr_un` of Solaris to the one on AIX:

```
/*
 * Definitions for UNIX IPC domain.
 */
struct  sockaddr_un {
        u_char  sun_len;                /* sockaddr len including null */
        u_char  sun_family;             /* AF_UNIX */
        char    sun_path[108];          /* path name (gag) */
};

/*
 * Definitions for UNIX IPC domain.
 */
struct  sockaddr_un {
        short   sun_family;             /* AF_UNIX */
        char    sun_path[108]           /* path name (gag) */
};
```

Figure 42. Sockaddr_un. The first structure is from AIX and the second from Solaris (SunOS 5.3).

Vanilla UNIX is *without* the sun_len member, which was included as a part of 4.3BSD Reno and later. OS/400 is using the vanilla version.

Chapter 6. Development Environment on AS/400 System

This chapter includes the topics of the applications development environment of the AS/400 system from the viewpoint of UNIX C applications porting. We expect you are already familiar with the general description of the AS/400 system′s applications development environment, especially for C applications such as ADT/400 and ILE C/400. For those who need basic understandings of it, refer to the Appendix C, "Development Cycle of ILE C/400 Applications" on page 259.

6.1 Editors and Programs Location

Probably the first thing we need to remind you of is the location of editors and created objects (programs, if you will) on the AS/400 system. If you have read Chapter 3, "File System - AS/400 Integrated File System" on page 25, you know that the AS/400 system has a file system with the characteristics of UNIX file systems: they are /QOpenSys file system and / ″root″ file system. Unfortunately, the system editor, SEU, does not work on those file systems. In fact, SEU only works on DB files (that is, the native, traditional AS/400 file system: /QSYS.LIB file system).

This means you cannot edit stream files stored in either / ″root″ file system or /QOpenSys file system. This also means, therefore, you have to port your source files into /QSYS.LIB file system as members in the source physical files in your library. One reminder here is that there are only three levels of directory structure (well, four levels if you include QSYS.LIB itself) in the /QSYS.LIB file system. All of your source files should be in three level relationship of Library/SourcePhysicalFile/Member. Depending on the complexity of your applications structure, this could be a major concern.

Another point of consideration is that when you create modules, objects, or programs on the AS/400 system, their target locations should always be as modules (*MOD) or programs (*PGM) in the libraries. Again, the maximum level of directory structure here is limited to two (Library/Object) and you cannot store and execute them in file systems other than the /QSYS.LIB file system, (such as / ″root″ file system or /QOpenSys file system).

One possible solution is having a cross-compiler generally available where you develop the sources in UNIX platforms, such as AIX, and then create the objects on the AS/400 system. At the moment of writing this book, there is no such things generally available.

Of course another solution, which is more fundamental, is having the system editor work on any files on any file system of the AS/400 system and having an executables interpreter on the AS/400 system, but it sounds like a long shut at this moment.

6.2 ILE C/400 Compiler

The ILE C/400 Compiler is a full standard ANSI C compiler for the AS/400 system. It provides the System Application Architecture (SAA) C level 2 interfaces in addition to the ANSI C standards. This section introduces ILE/C 400 compiler specifics only because its language element is compliant to ANSI C.

6.2.1 Packed Qualifier

The **_Packed** Qualifier removes padding between members of structures and unions. However, the storage saved using packed structures and unions may come at the expense of runtime performance. Most machines access data more efficiently if it is aligned on appropriate boundaries. With packed structures and unions, members are generally not aligned on natural boundaries, and the member assessing operations (dot (&dot). and -> operator) are slower.

But pointers are always aligned on their natural boundaries, 16 bytes, even in **_Packed** structures and unions.

The Figure 43 on page 159 shows the differences between the normal structure and the packed structure.

```
  struct a {                  _Packed struct b {
      char    ch;                   char    ch;
      double  d;                    double  d;
      void   *p;                    void   *p;
 };                           };

                   struct a                     _Packed struct b
  Adress: 0     :     ch      Adress: 0     :     ch
          1-7   :     -               1-8   :     d
          8-15  :     d               9-15  :     -
          16-31 :     p               16-31 :     p
```

Figure 43. Storage Alignment

6.2.2 Special Type

The ILE/C compiler provides the type of decimal, **Packed Decimal**, besides the standard types, such as char, int, double, and so on. To declare this data type, the header file **<decimal.h>** must be included in the source code. Its usage is:

```
    decimal(n,p)  varid;
```

where n is number of digits and p is decimal points. The n and p have ranges of p • n, 1 • n • 31, and 0 • p. The size of packed decimal is (n+1)/2.

6.2.3 Macros Defined Only by ILE/C Compiler

The C language provides some predefined macros, such as __FILE__, __LINE__, __DATE__, __STDC__, and so on. Besides these, the ILE/C compiler provides more macros, which are dependent on ILE environment. The predefined macros only by ILE/C compiler are:

__ILEC400__
: You can use this macro in your source code that is compiled for several platforms to block off code that is to be compiled only for the the AS/400 platform with #ifdef __ILEC400__ or #if defined (__ILEC400__) preprocessor directives.

__TIMESTAMP__
: A character string literal containing the date and time when the source file was last modified. The date and time is in the form:

```
                     ″Day Mmm dd hh:mm:ss yyyy″:
```

.

_IS_QSYSINC_INSTALLED
This macro is defined when the QSYSINC library is successfully added to the product portion of the library list.

6.2.4 Include Directive

A preprocessor include directive causes the preprocessor to replace the directive with the contents of the specified file. Because the file structure of AS/400 system is different from other systems, the search path and the format of the include file is different from the UNIX system.

The Table 15 shows the search path and the format of include files.

Table 15. Search Paths for #include Directive Used by ILE C/400

Include Type	**Search Path**
<member>	QCSRC
″member″	The SRCFILE of the root source member.
<file/member>	Searches the current library list (*LIBL).
″file/member″	1. Check the library containing the root source member. 2. Searches the user portion of the library list. 3. Searches the library list.
<lib/file/member>	Searches for lib/file/member only.
″lib/file/member″	1. Searches for lib/file/member only 2. Searches the user portion of the library list.

6.2.5 ILE C/400 Specific #pragma Preprocessing Directives

The table Table 16 on page 161 shows the ILE C/400 specific #pragma preprocessing directives. For the syntax and usage, refer to the ILE C/400 reference manual.

Table 16 (Page 1 of 2). #pragma Preprocessor Directives

Directive	**Description**
Argument	Specifies the argument-passing mechanism to be used for the procedure or typedef named as the first parameter.
Cancel_handler	Specifies that the function named is to be enabled as a user-defined ILE cancel handler at the point in the code where the #pragma cancel_ handler directive is located.
Checkout	Specifies whether or not the compiler should give informational message indicating possible programming errors when a CHECKOUT option other than *NONE is specified on the CRTCMOD or CRTBNDC command.
Convert	Specifies Code Character Set Identifier (CCSID) to use for converting the string literals from that point onward in a source file during compilation.
Disable_handler	Specifies the handler most recently enabled by either the exception_handler or cancel_handler pragma.
Exception_handler	Enables a user defined ILE exception handler at the point in the code where the #pragma exception_handler is located.
Inline	Specifies that function_name is to be inlined.
Linkage	Specifies that the external program is called using OS/400 calling conventions.
mapinc	Indicates that external AS/400 file descriptions are to be included in an ILE C/400 module.
Margins	Specifies the left and right margins to be used as the first and last column respectively, when scanning the records of the source member where the # pragma directive occurs.
Noinline	Specifies that a function will not be inlined.
Nomargins	Specifies that the entire input record is to be scanned for input.
Nosequence	Specifies that the input record does not contain sequence numbers.
Nosigtrunc	Specifies that no exception is generated at runtime when overflow occurs with packed decimals in arithmetic operation, casting, initialization, or function calls.

Table 16 (Page 2 of 2). #pragma Preprocessor Directives

Directive	Description
Descriptor	The #pragma descriptor directive is used to identify functions whose arguments have operational descriptors. It is useful when passing arguments to functions written in other languages that may have a different definition on the data types of the arguments.
Pointer	Allows the use of the AS/400 pointer types, such as space pointer, system pointer, invocation pointer, label pointer, suspend pointer, and open pointer.
Sequence	Specifies the columns of the input record that are to contain sequence numbers.

6.3 Shell Scripts versus CLP

In the UNIX system, there is a command interpreter, **sh**, which gets the input from the keyboard, parses the input, and finally executes.

A shell script is a file that contains sequences of shell commands just as you type them from the keyboard. Shell scripts enable you to customize your environment by adding your own commands. Shell provides also the language element with which the user can branch, or executes in a loop and so on. Even the output of one command can be redirected to the input of another command. For the UNIX system developer, the shell programming is a very powerful and flexible development tool and its script is a part of an application.

On the the AS/400 system there is a **control language (CL)**, which is the primary interface to the operating system. A single control language statement is called a **command**. For all commands, the operating system provides prompting support, default values for parameters, and validity checking. A command includes:

- Command name
- Command processing program (CPP)
- Parameters and values that are valid for the command
- Validity checking information

- Prompt text
- Online help information

Similar to the shell script in the UNIX system, a sequence of commands can be written in a source member. But to execute this file, it should be successfully compiled.

6.3.1 CL Programming

A CL procedure is a group of CL commands that tells the system where to get input, how to process it, and where to place the result. When you enter CL commands individually (from the Command Entry display or command line), each command is separately processed. When you enter CL commands as source statements for a CL procedure, this can be compiled into a program to be run. The source can be compiled into a module. To create a CL program, the following steps are required.

1. Source creation: CL procedures consist of CL commands. In most cases, source statements are entered into a database file in the logical sequence determined by your application design.
2. Module creation: Using the Create Control Language Module (CRTCLMOD) command, this source is used to create a system object. The created CL module can be bound into programs. A CL module consists one CL procedure. Other HLL languages may contain multiple procedures for each module.
3. Program creation: Using the Create Program (CRTPGM) command, this module is used to create a program.

Notes:

If you want to create a program consisting of only one CL module, you can use the Create Bound CL Program (CRTBNDCL) command, which combines steps 2 and 3.

The following Table 17 shows the parts of CL procedures.

Table 17 (Page 1 of 2). Parts of a CL Procedure

Command	**Keyword**	**Description**
PGM	PGM PARM (&A)	Optional PGM command beginning the procedure and identifying any parameters received.

Table 17 (Page 2 of 2). Parts of a CL Procedure

Command	Keyword	Description
Declare	DCL, DCLF	Mandatory declaration of procedure variables when variables are used. The declare commands must precede all other commands except the PGM command.
CL processing	CHGVAR, SNDPGMMSG, OVRDBF, DLTF,...	CL commands used as source statements to manipulate constants or variables (this is a partial list).
Logic control	IF, THEN, ELSE, DO, ENDDO, GOTO	Commands used to control processing within the CL procedure.
Functions	%SBUSTR(%SST), %SWITCH, %BINARY(%BIN)	Built-in functions and operators used in arithmetic, relational, or logical expressions.
Program control	CALL,RETURN	CL commands used to pass control to other programs.
Procedure control	CALLPRC,RETURN	CL commands used to pass control to other procedures.
ENDPGM	ENDPGM	Optional End Program command.

6.3.2 Creating a CL Program

If a CL source code editing is finished with the command STRSEU, it can be compiled into module or direct into a program.

Note: The editor provides the user with a CL command prompt for the syntax help, online help for the command, and validity checking for command parameters.

The related CL program creation commands are listed in Table 18.

Table 18 (Page 1 of 2). CL Program Creation Commands

Command	Description
CRTCLMOD	Creates a CL module.

Table 18 (Page 2 of 2). CL Program Creation Commands

Command	**Description**
DLTMOD	Deletes a module.
DLTPGM	Deletes a program.
CRTBNDCL	Create a bound CL program.
CRTPGM	Creates a program.
CRTSRVPGM	Creates a service program.
CRTCLPGM (*)	Creates a CL program.

Note:

The command CRTCLPGM creates an OPM program.

6.3.3 CL Programs for Shell Scripts Examples

This section provides some CL programs examples for certain shell scripts.

```
#  create a object with debug information
#  source file name must be passed without extension.

cc -c -o $1 -g $dollar.c
```

If the preceding shell script is translated into the CL source program, it is shown as follows.

```
/* This program creates a module from the source member, which */
/* will be passed as parameter. It is assumed the creation      */
/* library is the current library, which will be set with the   */
/* command CHGCURLIB libname.                                   */

  PGM        PARM(&MBRNAME)
  DCL        VAR(&MBRNAME)  TYPE(*CHAR) LEN(10)
  CRTCMOD    MODULE(&MBRNAME) SRCFILE(*CURLIB/QCSRC)     +
               OUTPUT(*PRINT) DBGVIEW(*ALL)
  ENDPGM
```

If modules are successfully compiled, they should be linked into a program. In the unix shell, script is shown as follows.

```
# This script binds objects to the program cprog.

ld -o cprog cmain.o ctable.o cinput.o cpwd.o cls.o ccd.o ccat.o
```

The translated CL source is shown as follows.

```
/* This CL program binds C modules to the program cprog. */
/* The default library is *CURLIB                         */

PGM
CRTPGM PGM(CPROG) MODULE(CMAIN CINPUT CTABLE CCD CPWD CCAT CLS)
ENDPGM
```

or

```
/* This CL program binds C modules and service programs  */
/* to the program cprog.                                 */
/* The default library is *CURLIB                        */

PGM
CRTPGM PGM(CPROG) MODULE(CMAIN CINPUT CTABLE)     +
       BNDSRVPGM(CCD CPWD CCAT CLS)
ENDPGM
```

Another example is an utility program, which reads the symbol names from the service program or programs. It is useful if you want to know where the symbol is defined or to find the symbols.

```
             PGM        PARM(&PROG &LIB)
             DCL        VAR(&PROG) TYPE(*CHAR) LEN(10)
             DCL        VAR(&LIB) TYPE(*CHAR) LEN(10)
             DCLF       FILE(QSYS/QADSPOBJ)

             CHGJOB     LOG(*SAME *SAME *MSG)
             CLRPFM     FILE(MYLIB/EXPINFO)
             MONMSG     MSGID(CPF3142) EXEC(CRTPF +
                          FILE(MYLIB/EXPINFO) RCDLEN(132))
   /*                                                      */
             DSPOBJD    OBJ(&LIB/&PROG) OBJTYPE(*SRVPGM) +
                          OUTPUT(*OUTFILE) OUTFILE(QTEMP/TEMPFIL)

             OVRDBF     FILE(QADSPOBJ) TOFILE(QTEMP/TEMPFIL)
 LOOP:       RCVF
             MONMSG     MSGID(CPF0864) EXEC(GOTO CMDLBL(END))
             DSPSRVPGM  SRVPGM(&ODLBNM/&ODOBNM) OUTPUT(*PRINT) +
                          DETAIL(*PROCEXP)
             SNDPGMMSG  MSG('Processing file' *bcat &ODLBNM *cat +
                          '/'  *cat &ODOBNM *cat '.')
             CPYSPLF    FILE(QSYSPRT) TOFILE(MYLIB/EXPINFO) +
                          SPLNBR(*ONLY) MBROPT(*ADD)
             MONMSG     MSGID(CPF3303) EXEC(GOTO CMDLBL(END))
             DLTSPLF    FILE(QSYSPRT) SPLNBR(*ONLY)
  /*                                                         */
             GOTO       CMDLBL(LOOP)
  /*                                                         */
 END:        ENDPGM
```

If shell scripts are part of an application, they should be rewritten in CL but just as a reminder, CL programs have the following benefits:

- Very fast
- Bindable
- Can be called as an external function
- Good interface with other programs

6.4 Makefile

The ILE C/400 provides also a make utility **TMKMAKE**. But the usage of it is a little different from one on the UNIX system. If you want use the make utility, you must first create it. This is done by creating a CL program that builds the objects you need. The complete source codes of TMKMAKE are available for the user and are in the QATTSYSC file in the QUSRTOOL library.

6.4.1 How to Create Make Utility TMKMAKE

You must consider first where you want to create the utility. If you want it in the MYLIB library, for example, enter the following command.

```
CRTCLPGM PGM(MYLIB/TMKINST) SRCFILE(QUSRTOOL/QATTCL)
```

The library MYLIB must already exist.

Now you can run the install program you have just created in the library MYLIB. If you enter the following command, the TMKMAKE tool is installed.

```
CALL MYLIB/TMKINST MYLIB
```

The library name can be different from the library in which you have created the install program **TMKINST**.

6.4.2 Make Utility Example

To create the target program CPROG, enter the following command:.

```
TMKMAKE SRCFILE(SEY/QCSRC) SRCMBR(MAKEFILE)
```

The name of library, source file, and source member are used here, for example.

```
# This makefile is used to create CPROG

#
# The silent command instructs TMKMAKE not to echo the statements
#

.SILENT:

#
# define macros for the commands and options to be used
#

CCOPTS=OUTPUT(*PRINT) DBGVIEW(*ALL) REPLACE(*YES)
CC=CRTCMOD
LD=CRTPGM
AR=CRTSRVPGM
LIB=SEY

#
# define macros which describe the nesting within the includes
#

commh   = ccomm.h
protoh  = crpoto.h
allhs   = $(commh) $(protoh)

#
# describe the rule the create the *PGM CPROG
#

CPROG<PGM>:CMAIN<MODULE> CINPUT<MODULE> CTABLE<MODULE> MYSRVPGM<SRVPGM>
        $(LD) PGM($(LIB)/CPROG) \
              MODULE($(LIB)/CMAIN $(LIB)/CINPUT $(LIB)/CTAB
LE) \
              BNDSRVPGM(MYSRVPGM)
```

```
#
# describe the rules to create the *SRVPGM objects
#

MYSRVPGM<SRVPGM>:CCD<MODULE> CPWD<MODULE> CCAT<MODULE> CLS<MODULE>
        $(AR) SRVPGM($(LIB)/MYSRVPGM) \
              MODULE($(LIB)/CCD $(LIB)/CPWD $(LIB)/CCAT &do
llar.(LIB)/CLS) \
              EXPORT(*ALL)


#
# describe the rules to create the *MODULE objects
#

CMAIN<MODULE>: CMAIN.QCSRC<FILE> $(ALLHS)
        $(CC) MODULE($(LIB)/CMAIN) SRCFILE($(LIB)/QCSRC) &d
ollar.(CCOPTS)

CINPUT<MODULE>: CINPUT.QCSRC<FILE> $(ALLHS)
        $(CC) MODULE($(LIB)/CINPUT) SRCFILE($(LIB)/QCSRC) &
dollar.(CCOPTS)

CTABLE<MODULE>: CTABLE.QCSRC<FILE> $(ALLHS)
        $(CC) MODULE($(LIB)/CTABLE) SRCFILE($(LIB)/QCSRC) &
dollar.(CCOPTS)

CCD<MODULE>: CCD.QCSRC<FILE> $(ALLHS)
        $(CC) MODULE($(LIB)/CCD) SRCFILE($(LIB)/QCSRC) &dol
lar.(CCOPTS)

CPWD<MODULE>: CPWD.QCSRC<FILE> $(ALLHS)
        $(CC) MODULE($(LIB)/CPWD) SRCFILE($(LIB)/QCSRC) &do
llar.(CCOPTS)

CCAT<MODULE>: CCAT.QCSRC<FILE> $(ALLHS)
        $(CC) MODULE($(LIB)/CCAT) SRCFILE($(LIB)/QCSRC) &do
llar.(CCOPTS)

CLS<MODULE>: CLS.QCSRC<FILE> $(ALLHS)
        $(CC) MODULE($(LIB)/CLS) SRCFILE($(LIB)/QCSRC) &dol
lar.(CCOPTS)
```

Figure 44. Makefile

For more information about TMKMAKE, see QATTINFO(TMKINFO) in the QATTSYSC library.

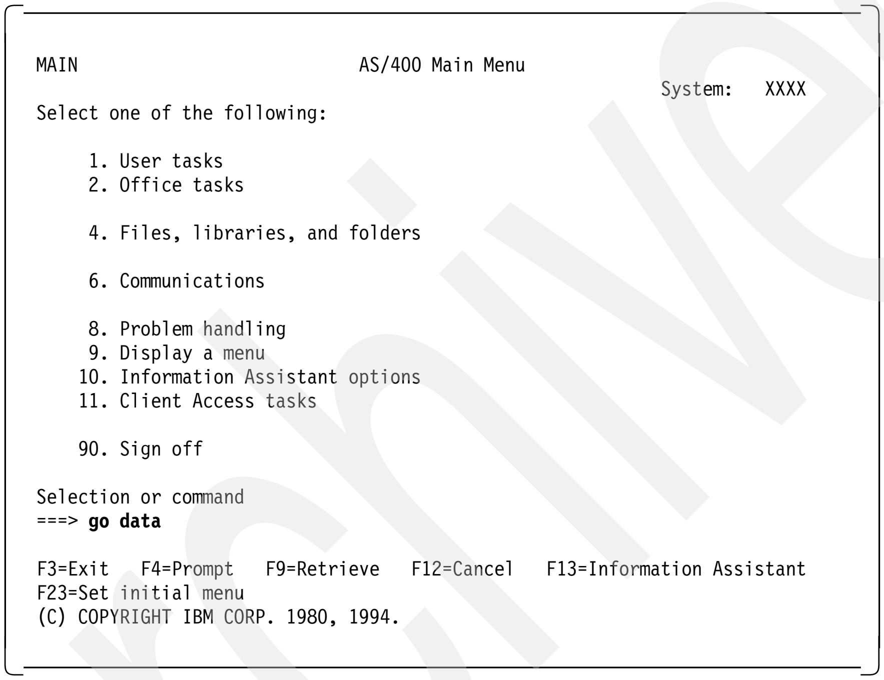
Appendix A. Integrated File System Tutorial

This appendix provides a tutorial for a short tour on the integrated file system. On the integrated file system menu, you can do the following operations by selecting options or command.

- Create and remove a directory.
- Display and change the name of the current directory.
- Add, display, and remove object links.
- Copy, move, and rename objects.
- Check out and check in objects.
- Save (back up) and restore objects.
- Display and change object owners and user authorities.
- Copy data between stream files and database file members.

A.1 Get into the Integrated File System

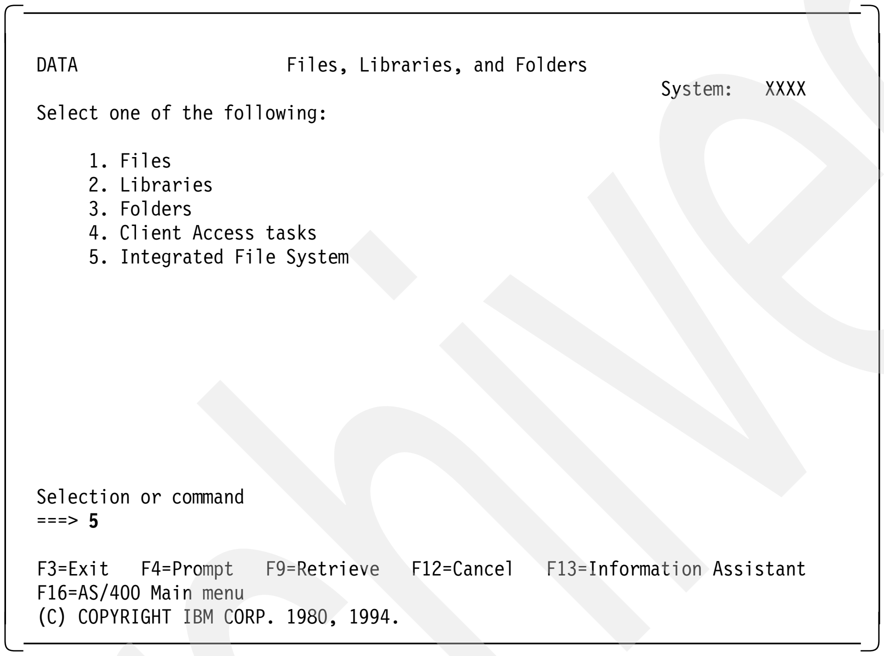
On any command line, enter **go data** as is shown on the following display.



```
MAIN                          AS/400 Main Menu
                                                            System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> go data

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
(C) COPYRIGHT IBM CORP. 1980, 1994.
```

Figure 45. AS/400 Main Menu with Go Data Command Specified

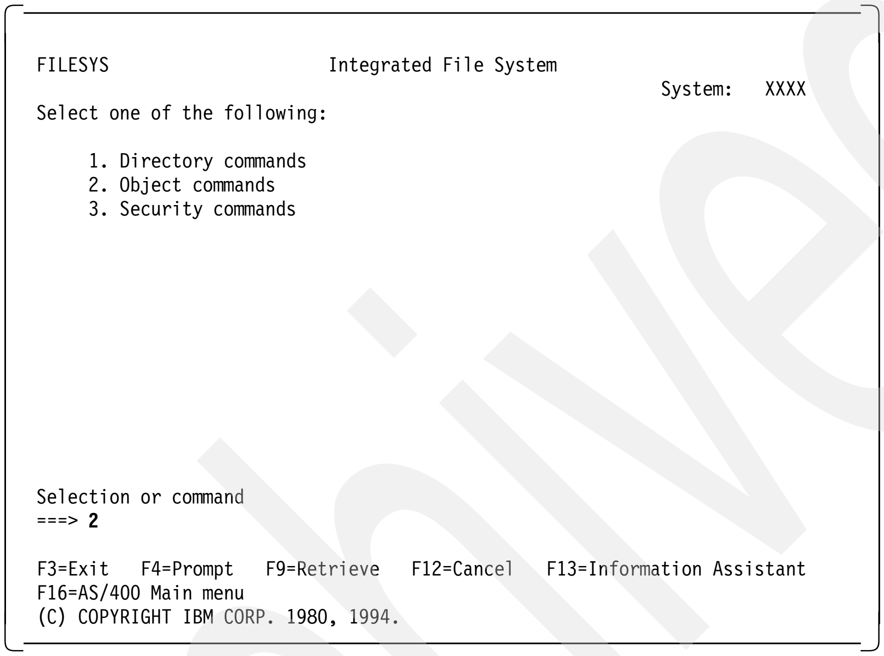
The Files, Libraries, and Folders menu is displayed.



```
DATA                    Files, Libraries, and Folders
                                                                    System:   XXXX
Select one of the following:

     1. Files
     2. Libraries
     3. Folders
     4. Client Access tasks
     5. Integrated File System









Selection or command
===> 5

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F16=AS/400 Main menu
(C) COPYRIGHT IBM CORP. 1980, 1994.
```

Figure 46. Files, Libraries, and Folders Menu with Option 5 Specified

Select 5 option and press Enter; then the integrated file system menu is displayed.



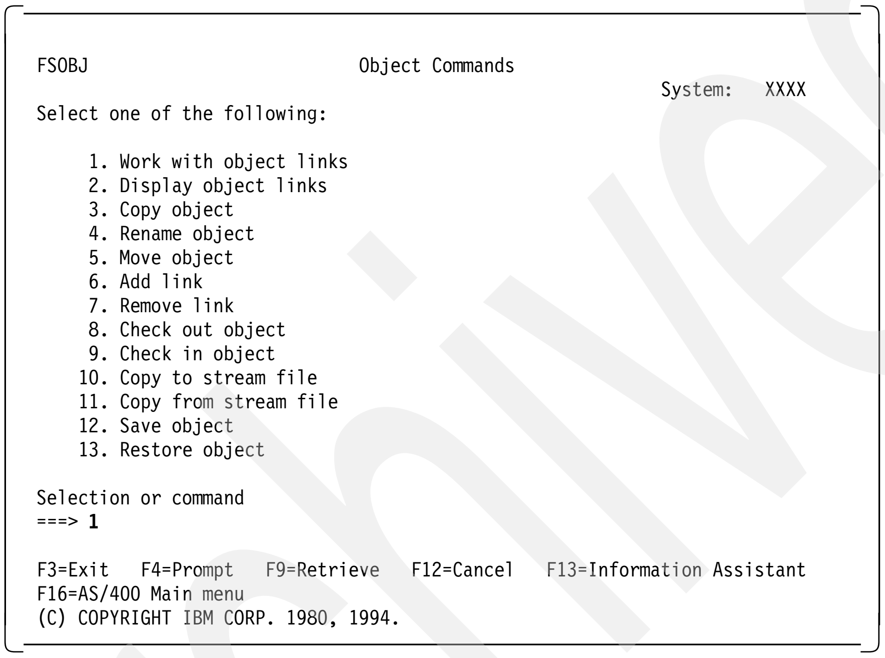
```
FILESYS                     Integrated File System
                                                                System:   XXXX
Select one of the following:

     1. Directory commands
     2. Object commands
     3. Security commands











Selection or command
===> 2

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F16=AS/400 Main menu
(C) COPYRIGHT IBM CORP. 1980, 1994.
```

Figure 47. Integrated File System Menu with Option 2 Specified

In this section, Directory commands and Object commands are explained on the Work with Object Links menu.

Let′s go to the Work with Object Links menu. Select the Object commands on the integrated file system menu.

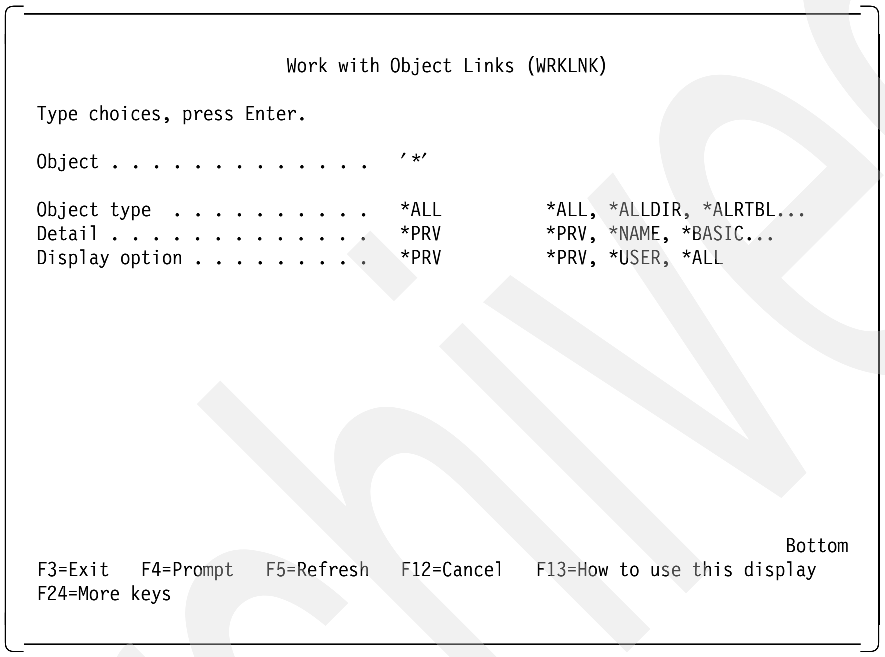
Select Work with object links option on Object Commands menu.



```
FSOBJ                          Object Commands
                                                               System:   XXXX
Select one of the following:

     1. Work with object links
     2. Display object links
     3. Copy object
     4. Rename object
     5. Move object
     6. Add link
     7. Remove link
     8. Check out object
     9. Check in object
    10. Copy to stream file
    11. Copy from stream file
    12. Save object
    13. Restore object

Selection or command
===> 1

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F16=AS/400 Main menu
(C) COPYRIGHT IBM CORP. 1980, 1994.
```

Figure 48. Object Commands Menu with Option 1 Specified

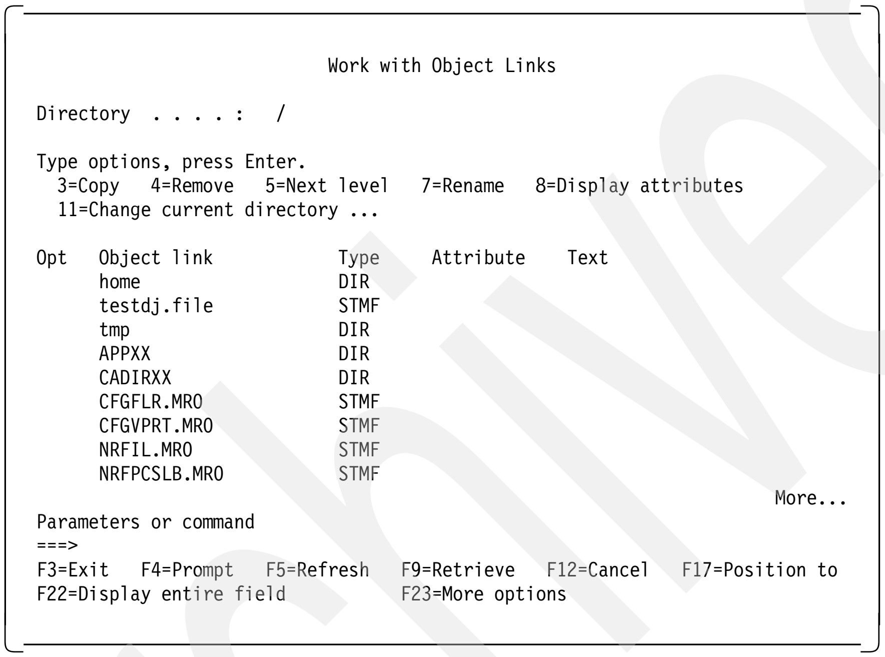
Press **Enter** on Work with Object Links Prompt text display.



```
                     Work with Object Links (WRKLNK)

 Type choices, press Enter.

 Object . . . . . . . . . . . . .   '*'

 Object type  . . . . . . . . . .   *ALL          *ALL, *ALLDIR, *ALRTBL...
 Detail . . . . . . . . . . . . .   *PRV          *PRV, *NAME, *BASIC...
 Display option . . . . . . . . .   *PRV          *PRV, *USER, *ALL










                                                                        Bottom
 F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
 F24=More keys
```

Figure 49. Work with Object Links Prompt Text

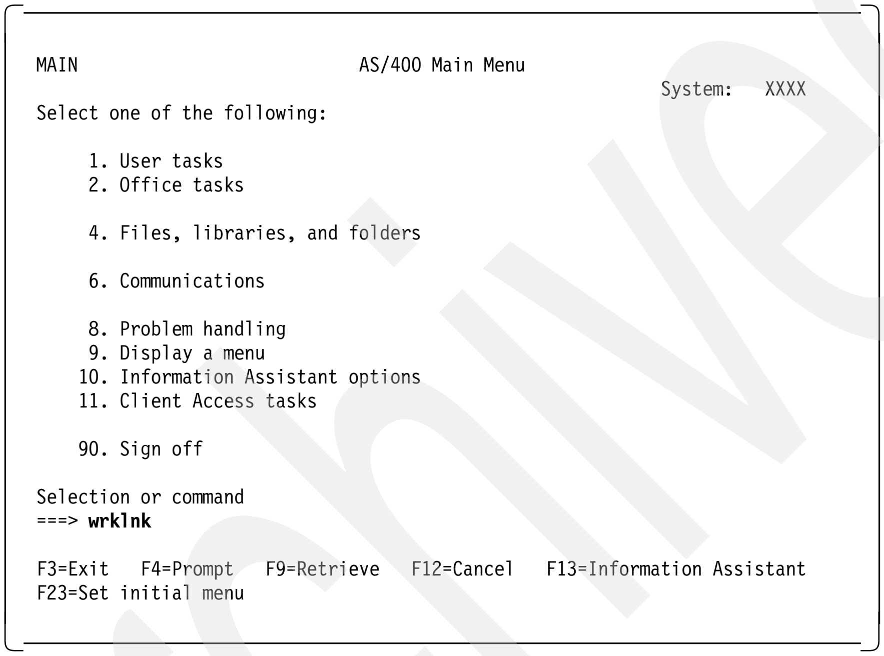
Then Work with Object Links menu is displayed.



```
                          Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type     Attribute    Text
      home                   DIR
      testdj.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
      NRFIL.MRO              STMF
      NRFPCSLB.MRO           STMF
                                                                       More...
Parameters or command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 50. Work with Object Links Menu

When you enter the **WRKLNK** command on the command line of any display, the Work with Object Link Menu is displayed also.



```
MAIN                            AS/400 Main Menu
                                                             System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> wrklnk

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
```

Figure 51. Work with Object Links Menu with WRKLNK Command Specified

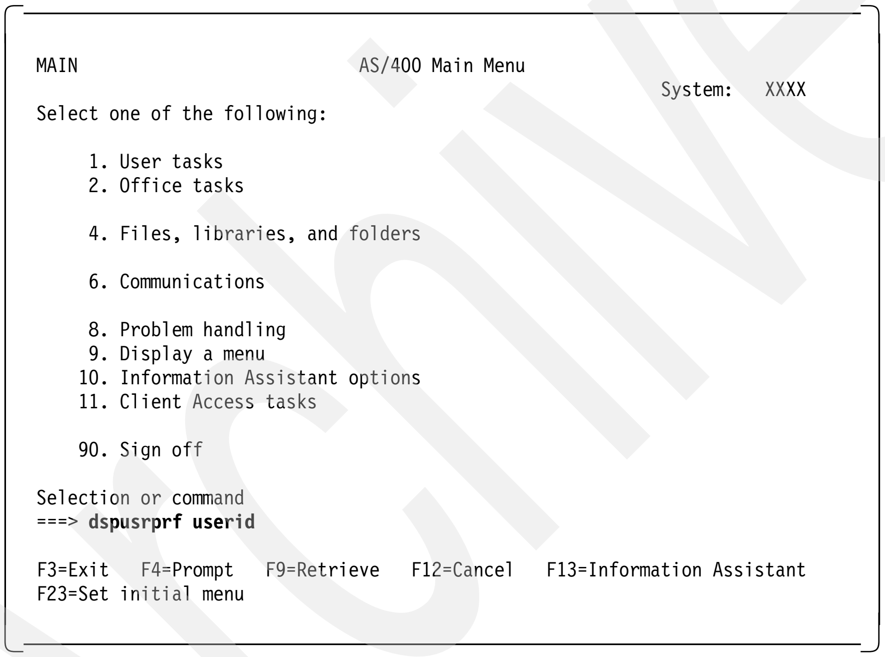
A.2 Current Directory and Home Directory

When a user requests an operation on an object such as a file, the system looks for the object in the user's **current directory** unless the user specifies a different directory path. The current directory is similar to the idea of the current library. It is also called the **current working directory** or just **working directory**.

The **home directory** is used as the current directory for a user when the user signs on the system. The name of the home directory is specified in the user profile for a user. When a job is started for a user, the system looks in the user profile for the name of the user's home directory. If a directory by that name does not exist on the system, the current directory is set to ′root′(/) directory.

Typically, the system administrator who creates the user profile for a user also creates the user's home directory. There is a subdirectory called *home* under the root directory that contains the home directory for each user. The system default is to use the name of the user profile for each user to identify the home directory for that user.

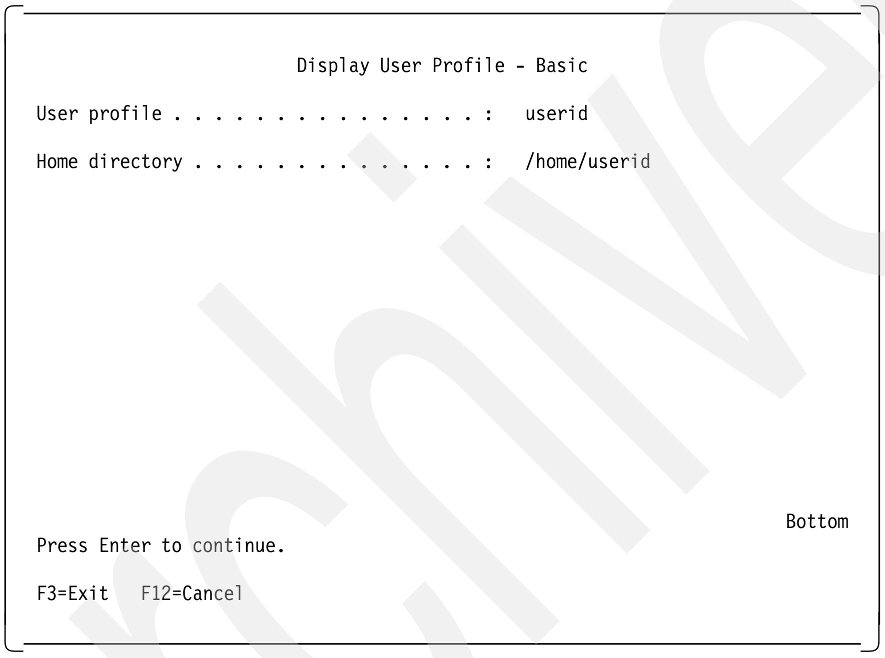
Let's see a home directory by using the DSPUSRPRF command.



```
MAIN                           AS/400 Main Menu
                                                             System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> dspusrprf userid

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
```

Figure 52. AS/400 Main Menu with DSPUSRPRF Command Specified

Press Enter; then the Display User Profile display is shown. This display shows you information about user profiles. You can reach the bottom of this information with the Page Down key and see the name of home directory on this display.



```
                         Display User Profile - Basic

 User profile . . . . . . . . . . . . . . . :   userid

 Home directory . . . . . . . . . . . . . . :   /home/userid













                                                                        Bottom
 Press Enter to continue.

 F3=Exit   F12=Cancel
```

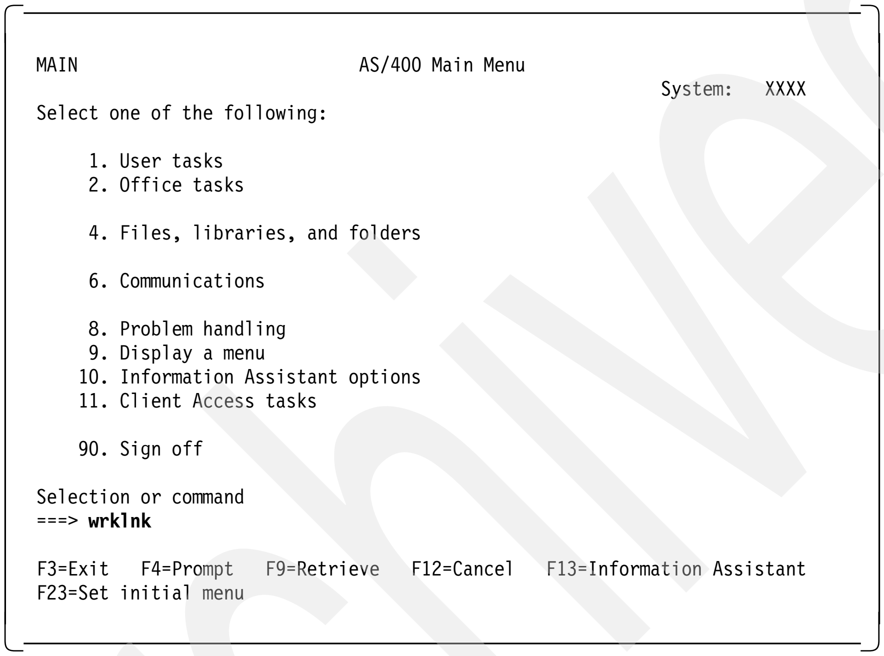
Figure 53. Display User Profile

Notes:

> This does not mean that the ′/home/userid′ directory is automatically created by the system when you sign on a system such as a UNIX system. This directory should be created manually.

So, let′s make the ″home″ directory. If your ″home″ directory is already created in your system, skip this procedure.

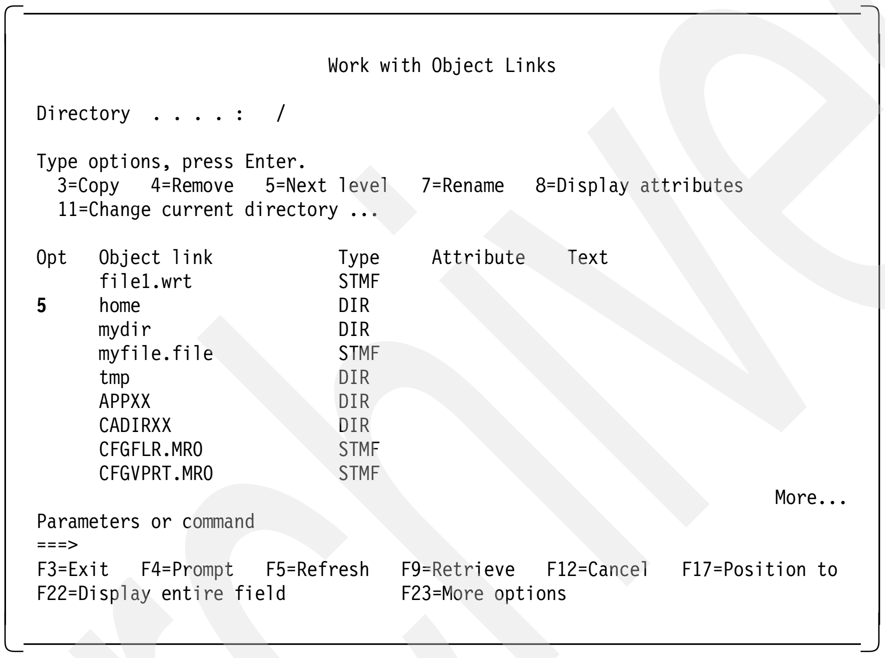
Enter WRKLNK on the command line of the MAIN menu.



```
MAIN                          AS/400 Main Menu
                                                            System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> wrklnk

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
```

Figure 54. AS/400 Main Menu with WRKLNK Command Specified

The ″/home/userid″ directory does not exist, so the ″root(/)″ directory is set to the current directory. Let′s get into the ″/home″ directory by typing **5** in the Opt column.

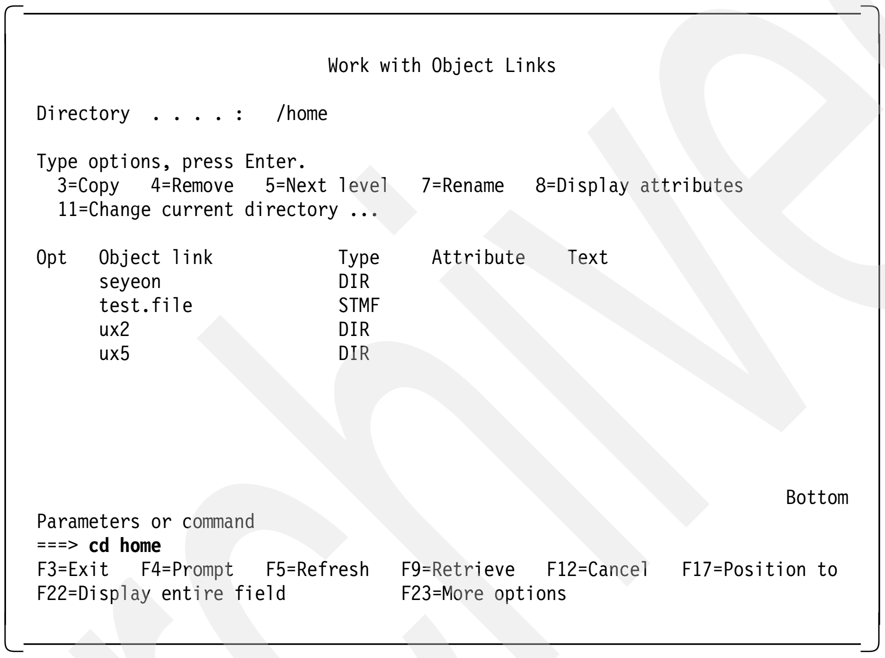


```
                          Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type     Attribute    Text
      file1.wrt              STMF
5     home                   DIR
      mydir                  DIR
      myfile.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
                                                                    More...
Parameters or command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 55. Work with Object Link Menu

You can confirm that ′/home/userid′ directory does not exist here.

Let's make a "userid" directory in the "/home" directory. To do this, we have to change the current directory to "/home" directory. You can change the current directory by using the "CD" command.

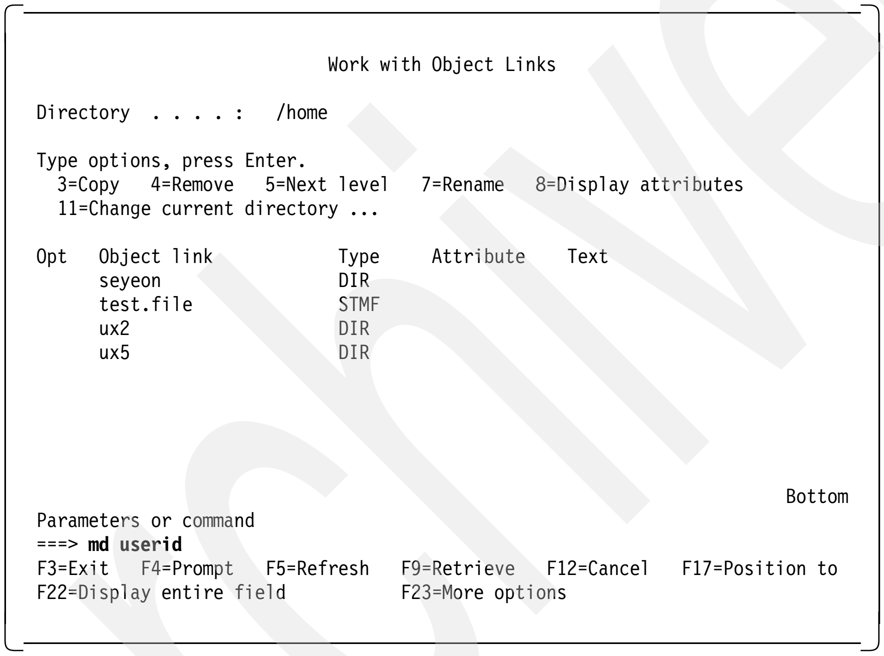


```
                           Work with Object Links

 Directory  . . . . :   /home

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link           Type     Attribute    Text
       seyeon                DIR
       test.file             STMF
       ux2                   DIR
       ux5                   DIR




                                                                   Bottom
 Parameters or command
 ===> cd home
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 56. Work with Object Link Menu with CD Command Specified

And then, you can make a ″userid″ directory by using the MD command. Type ′**md userid**′ and press Enter, you can see the ″Directory created″ message.

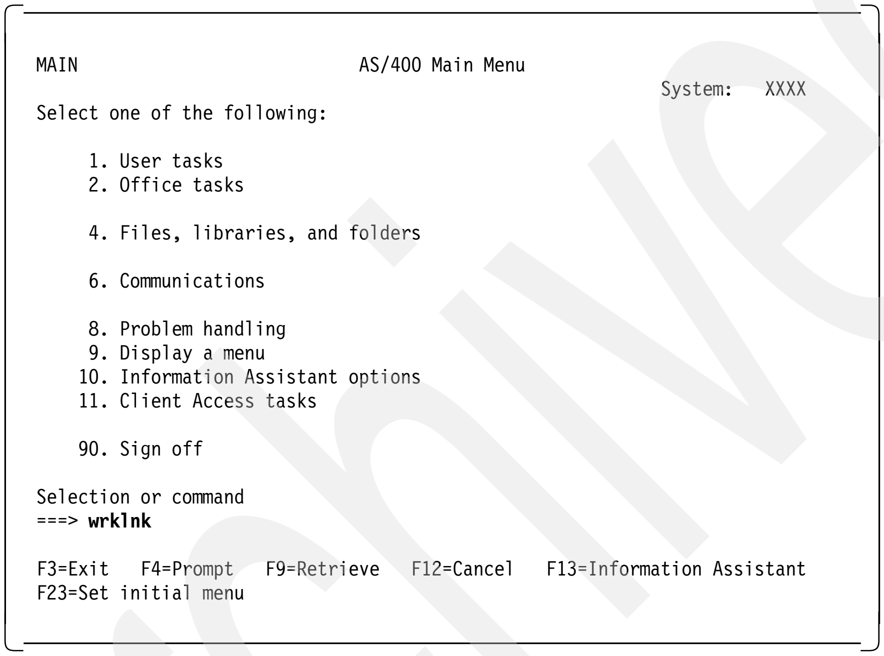
And you can get the refreshed list by pressing the <F5> key.



```
                              Work with Object Links

 Directory  . . . . :   /home

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       seyeon                 DIR
       test.file              STMF
       ux2                    DIR
       ux5                    DIR




                                                                        Bottom
 Parameters or command
 ===> md userid
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 57. Work with Object Link Menu with MD Command Specified

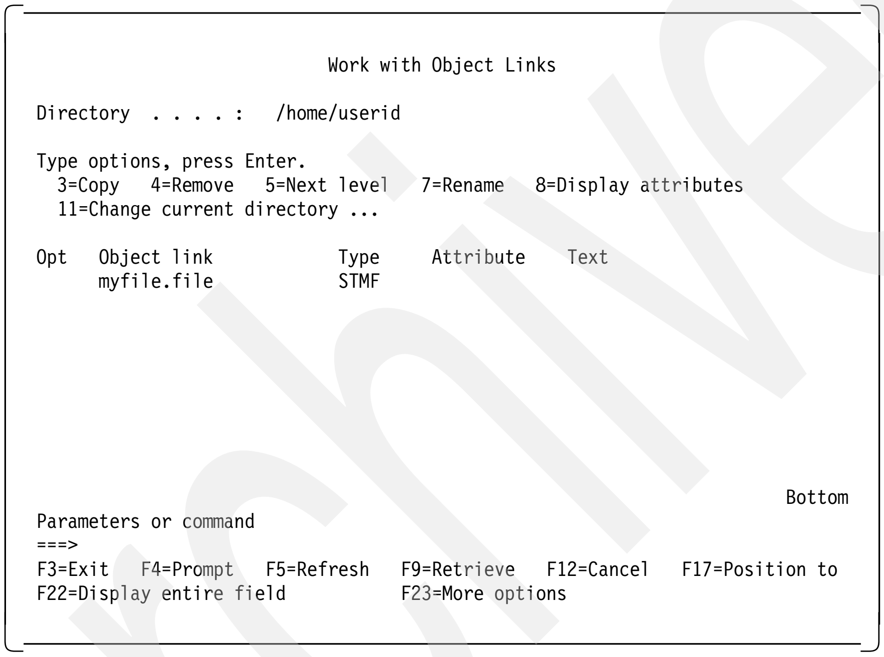
Now, sign off the system and sign on again to check the current directory. Enter WRKLNK on the command line of the MAIN menu.



```
MAIN                           AS/400 Main Menu
                                                               System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> wrklnk

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
```

Figure 58. AS/400 Main Menu with WRKLNK Command Specified

Press Enter; then the Work with Object Links menu for ″/home/userid″ is displayed. In the previous section, the Work with Object Links menu for the ″root″ directory was displayed because the ″/home/userid″ directory did not exist.



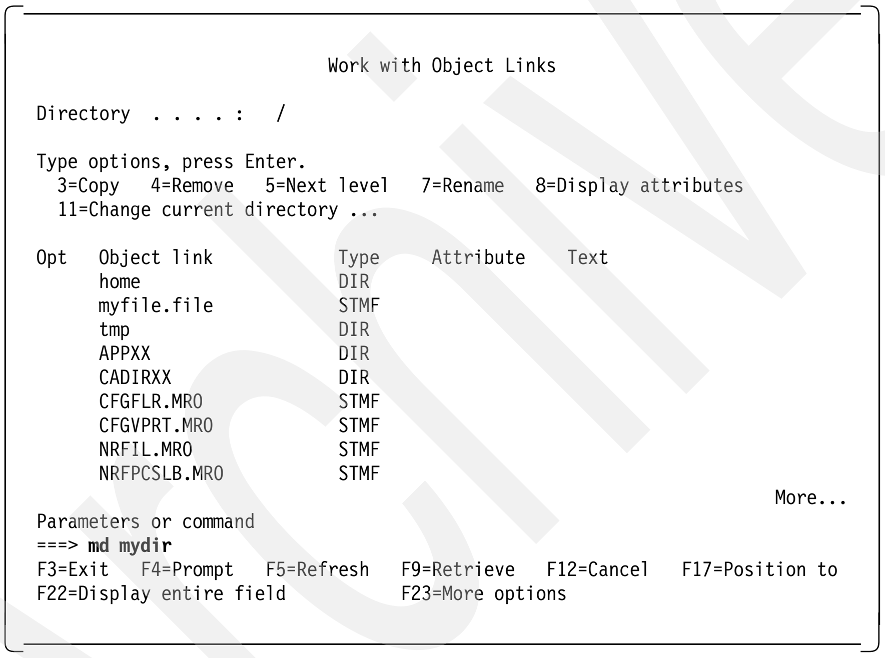
```
                              Work with Object Links

 Directory  . . . . :   /home/userid

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       myfile.file            STMF









                                                                         Bottom
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 59. Work with Object Links Menu

A.3 Create and Remove a Directory

Create a Directory: To create a directory, you can use the **MD**, the **MKDIR**, or the **CRTDIR** command.

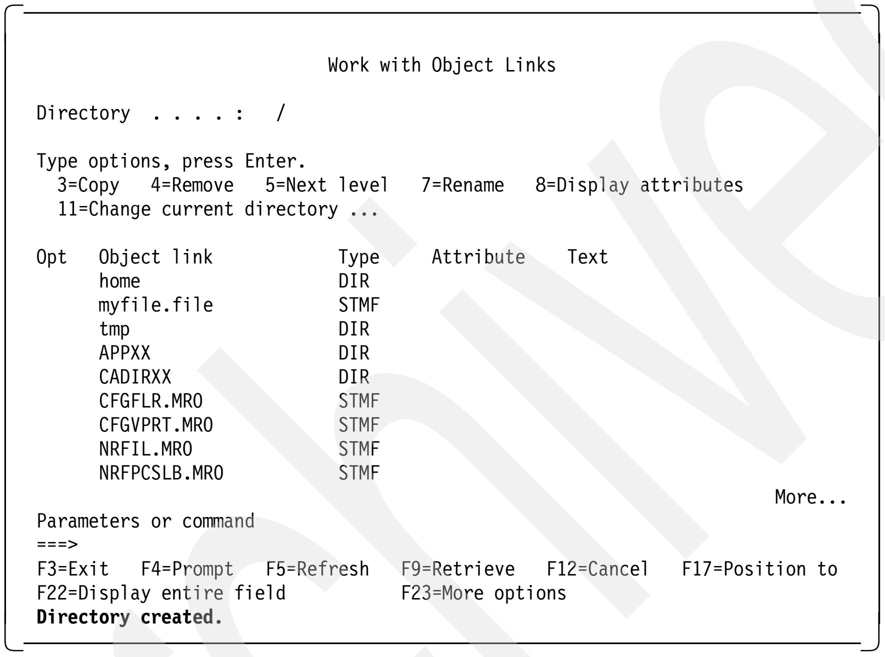
When you want to create a ″mydir″ directory, enter **MD mydir** on command line and press Enter.



```
                          Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type      Attribute    Text
      home                   DIR
      myfile.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
      NRFIL.MRO              STMF
      NRFPCSLB.MRO           STMF
                                                                   More...
Parameters or command
===> md mydir
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 60. Work with Object Menu with MD Command Specified

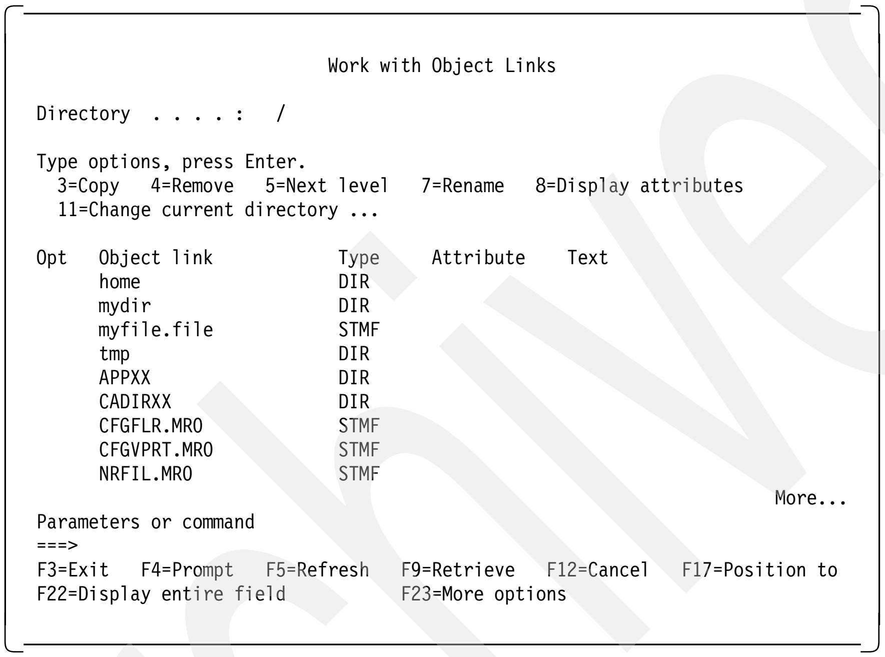
Then the Directory created message is displayed but there is no change on the list.



```
                          Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type     Attribute    Text
      home                   DIR
      myfile.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
      NRFIL.MRO              STMF
      NRFPCSLB.MRO           STMF
                                                                    More...
Parameters or command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
Directory created.
```

Figure 61. Work with Object Menu with Directory Created Message

When you press the F5 key, the list is updated.



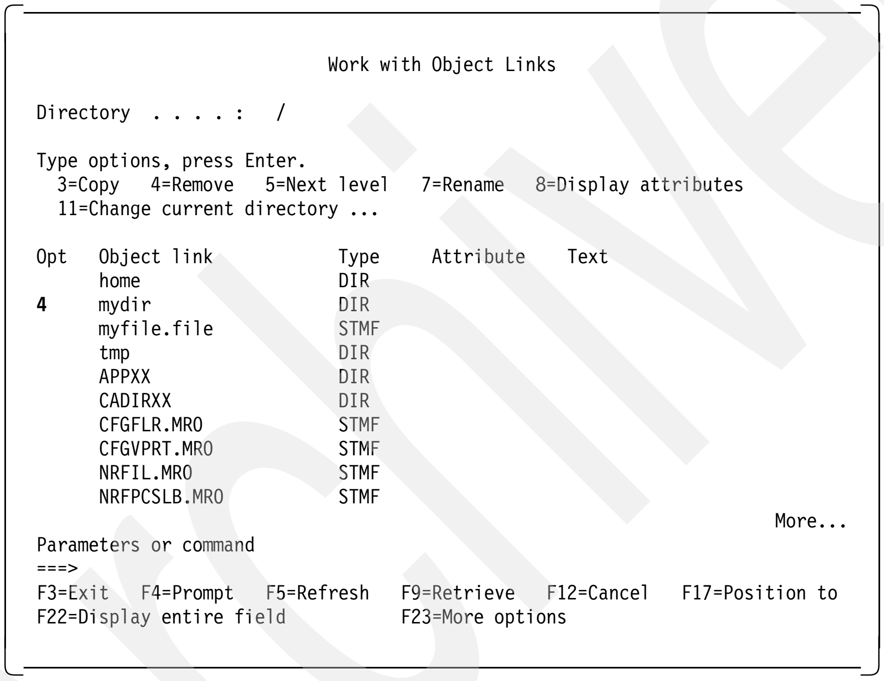
```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                        More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 62. Work with Object Menu with Refreshed List

MKDIR and CRTDIR commands also do the same work. So ″MKDIR mydir″ or ″CRTDIR mydir″ gives the same results as ″MD mydir″.

Remove a Directory: To remove a directory, you can use **option 4** or the**RD**, the **RMDIR**, or the **RMVDIR** command.

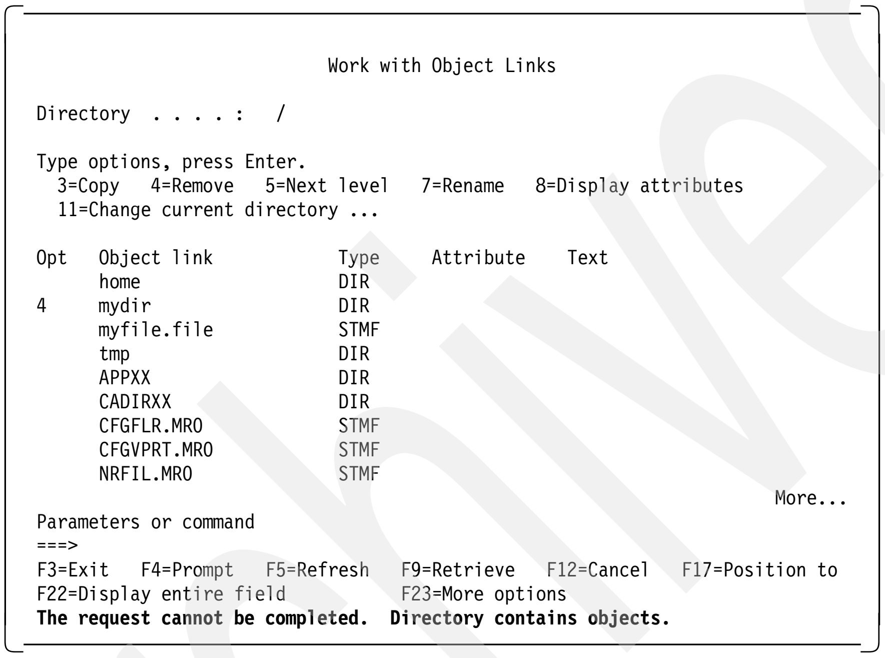
When you want to remove ″mydir″ directory, enter **2** in the Opt column or enter the **RD mydir** command on a command line and press Enter.



```
                             Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link           Type     Attribute    Text
       home                  DIR
 4     mydir                 DIR
       myfile.file           STMF
       tmp                   DIR
       APPXX                 DIR
       CADIRXX               DIR
       CFGFLR.MRO            STMF
       CFGVPRT.MRO           STMF
       NRFIL.MRO             STMF
       NRFPCSLB.MRO          STMF
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 63. Work with Object Menu with Option 4 Specified

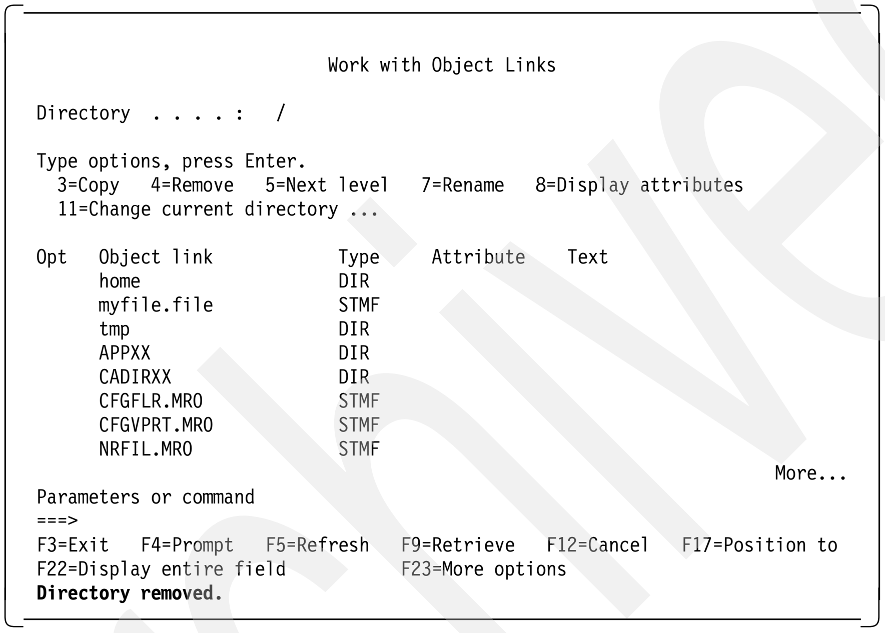
If ″mydir″ directory contains objects, then it cannot be erased.



```
                           Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link           Type     Attribute    Text
       home                  DIR
 4     mydir                 DIR
       myfile.file           STMF
       tmp                   DIR
       APPXX                 DIR
       CADIRXX               DIR
       CFGFLR.MRO            STMF
       CFGVPRT.MRO           STMF
       NRFIL.MRO             STMF
                                                                     More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 The request cannot be completed.  Directory contains objects.
```

Figure 64. Work with Object Menu with Rejection Message

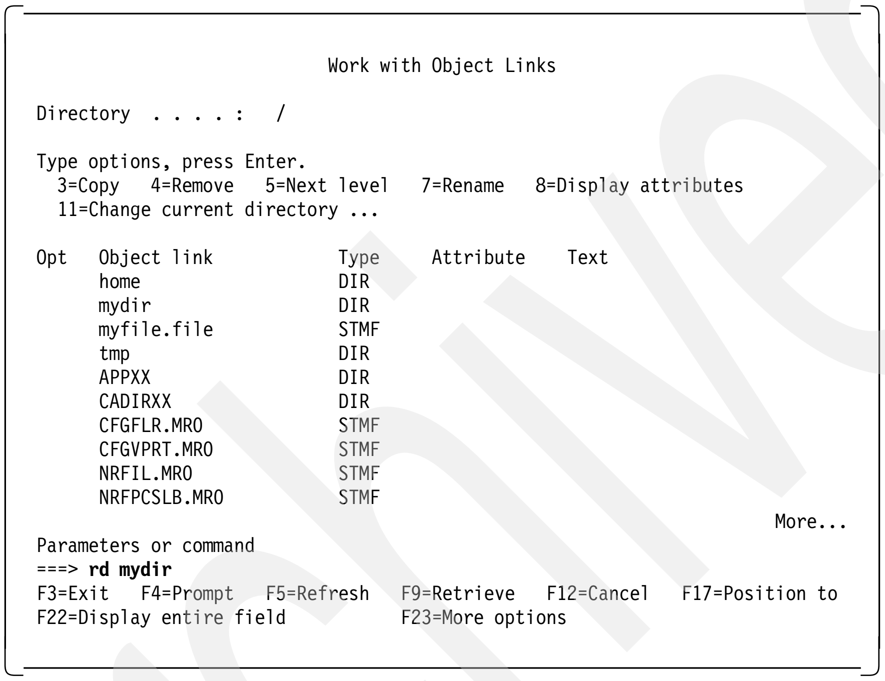
Or if ″mydir″ directory contains no objects, then it is erased through the confirmation window.



```
                           Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type     Attribute    Text
      home                   DIR
      myfile.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
      NRFIL.MRO              STMF
                                                                    More...
Parameters or command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
Directory removed.
```

Figure 65. Work with Object Menu with Directory Removed Message

Or, use the **RD mydir** command to remove a directory.



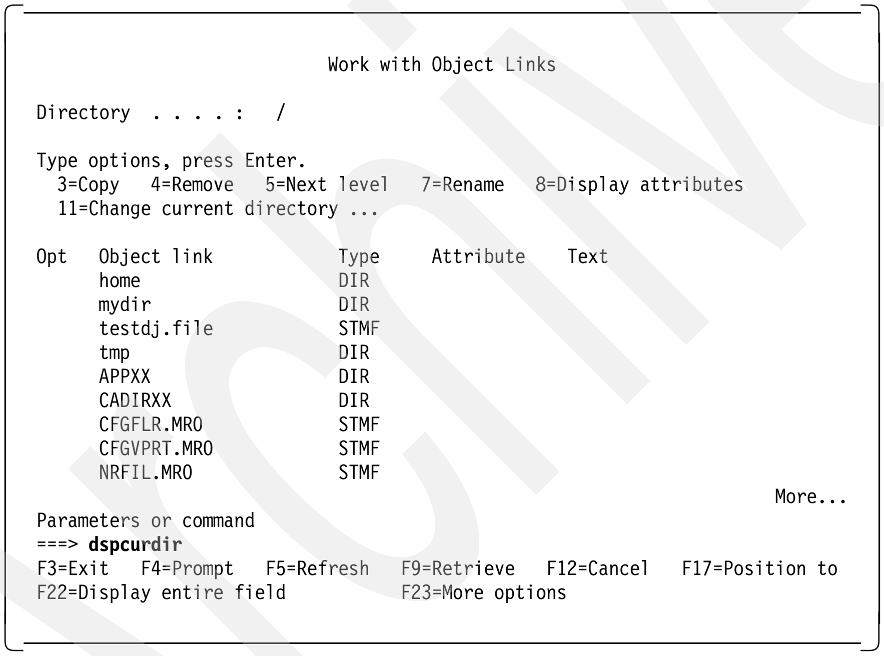
```
                              Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type      Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
       NRFPCSLB.MRO           STMF
                                                                       More...
 Parameters or command
 ===> rd mydir
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 66. Work with Object Menu with RD Command Specified

RMDIR and RMVDIR commands also do the same work. So ″RMDIR mydir″ or ″RMVDIR mydir″ have the same results as ″RD mydir″.

A.4 Display and Change Current Directory

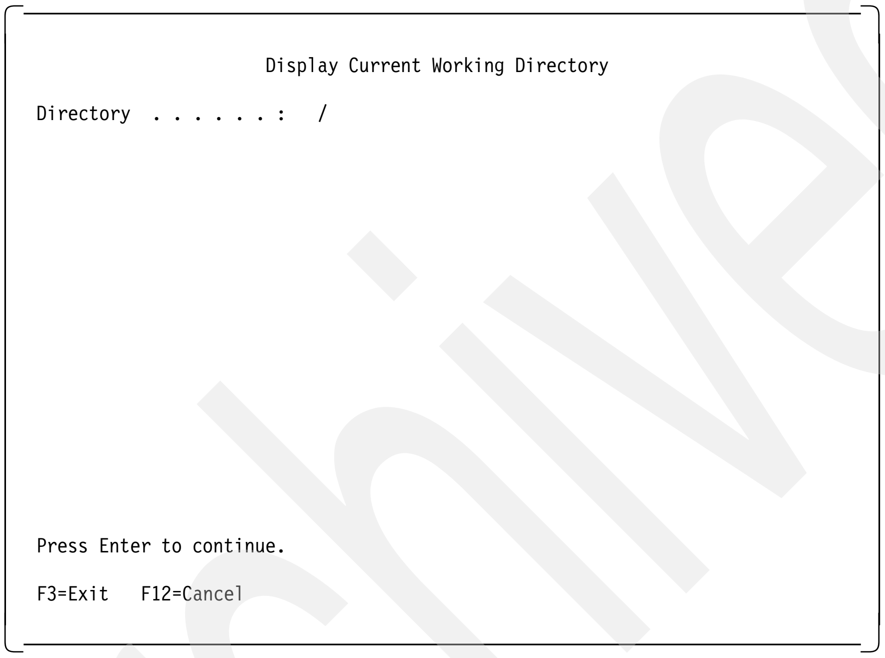
Display the Name of the Current Directory: When you want to know the name of current directory, type DSPCURDIR on a command line and press Enter.



```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       testdj.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                    More...
 Parameters or command
 ===> dspcurdir
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 67. Work with Object Menu with DSPCURDIR Command Specified

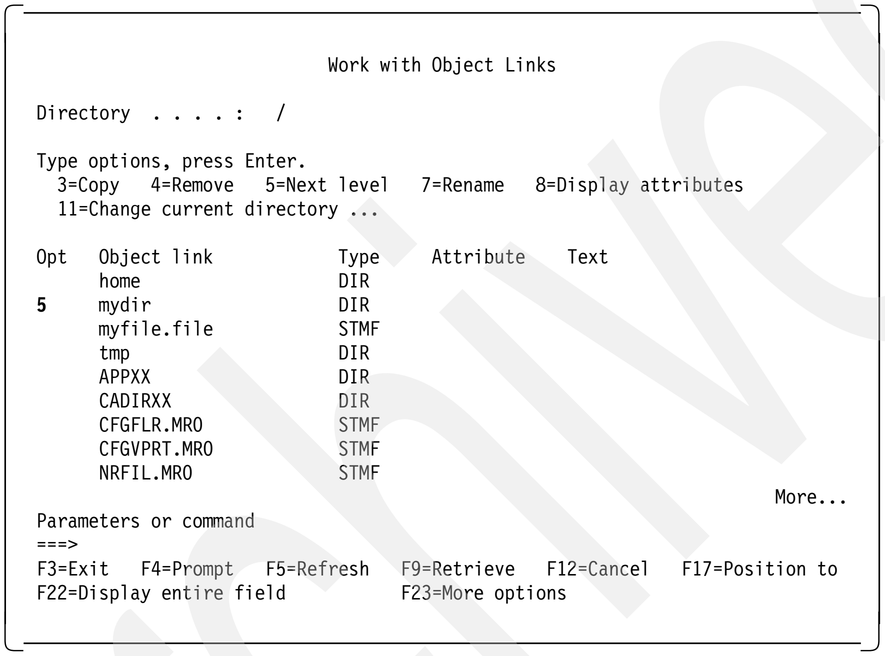
Then the Display Current Working Directory is displayed.



```
                      Display Current Working Directory

 Directory  . . . . . . :   /










 Press Enter to continue.

 F3=Exit   F12=Cancel
```

Figure 68. Display Current Working Directory

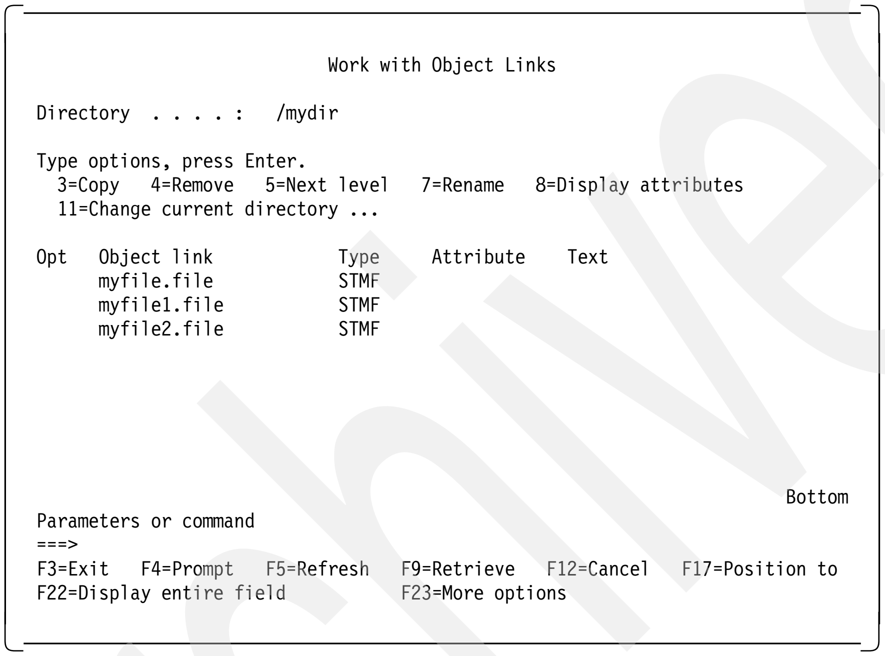
Work with next level: When you want to work with objects in ″mydir″ directory, then type **5** in the Opt column for ″mydir″ and press Enter.



```
                          Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
 5     mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 69. Work with Object Menu with Option 5 Specified

Then Work with Object Links for ″/mydir″ is displayed.



```
                             Work with Object Links

 Directory  . . . . :   /mydir

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       myfile.file            STMF
       myfile1.file           STMF
       myfile2.file           STMF







                                                                          Bottom
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 70. Work with Object Menu with Next Level

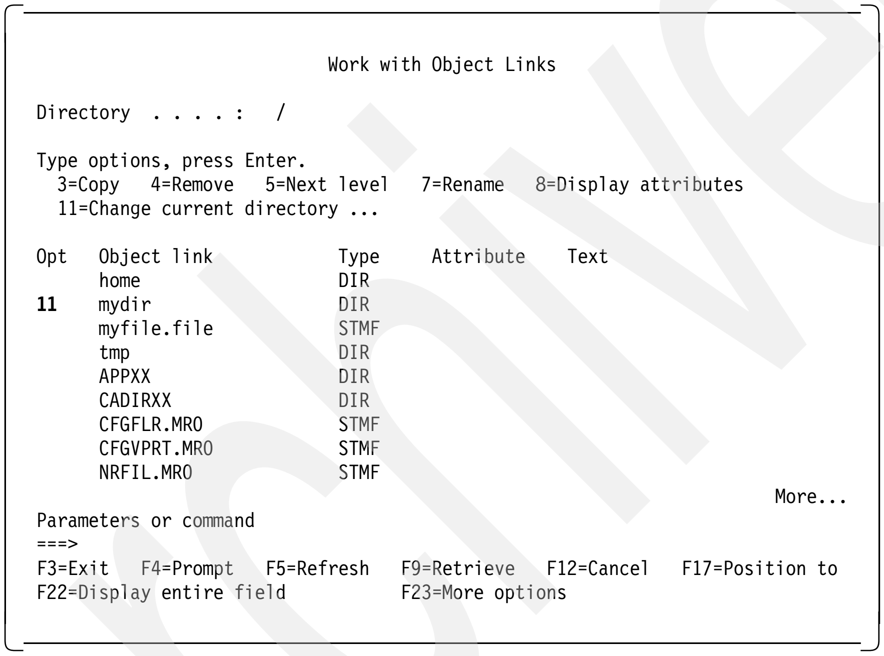
You can use this option to work with object links in a directory. You can use this option only on an object that is handled as a directory; for example, an object whose type is DIR(directory), DDIR(distributed directory), LIB(library), or FLR(folder).

In the UNIX system, if you change the directory with the command **cd mydir**, ″mydir″ is the current working directory. However, in the integrated file system, although you work in the directory ″/mydir″ in the preceding display, this is not the current working directory. This is just the same as the command **ls /mydir** in UNIX Because the current working directory is not set yet, the current working directory is still the ″root″ directory or home directory. So, if you copy (or move) an object in ″mydir″, this object is copied (or moved) to the root directory or home directory unless you specify another directory.

The following chapter shows you about changing the current directory.

Change Current Directory: To change the current directory, you can use Option **11**, the **CD**, or the **CHGCURDIR** command.

When you want to change the current directory to ″mydir″, then type **11** in the Opt column for the ″mydir″ object, or enter **cd mydir** on a command line.

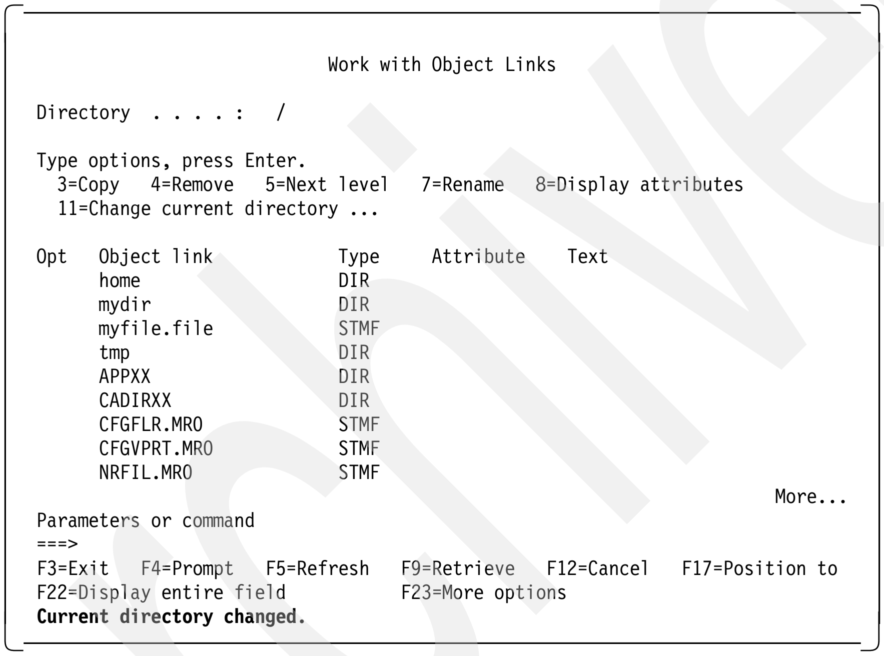


```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
 11    mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                     More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 71. Work with Object Menu with Option 11 Specified

Press Enter; then the Current directory changed message is displayed.

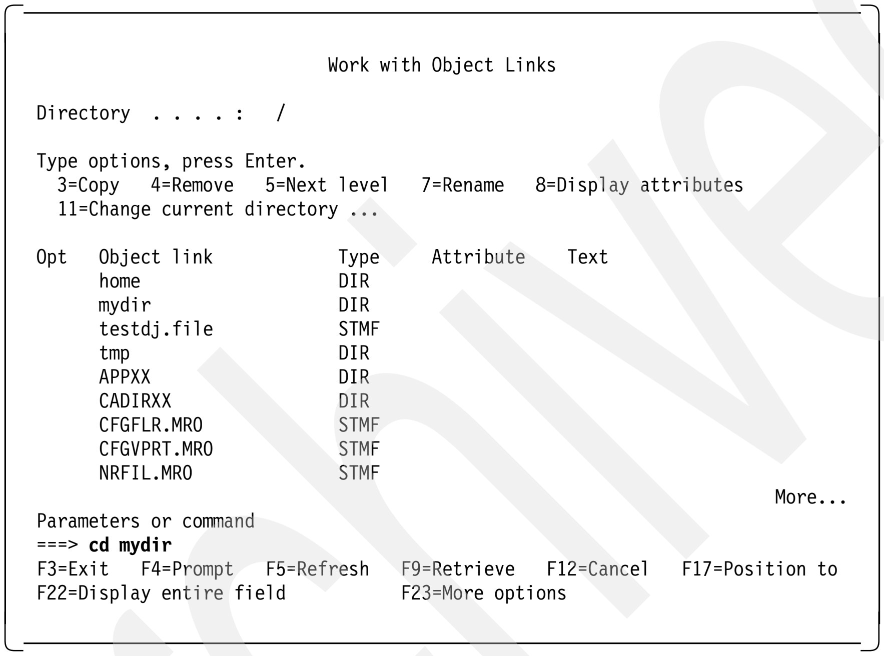
You can use option 11 only an object that is handled as a directory; for example, an object whose type is DIR(directory), DDIR(distributed directory), LIB(library), or FLR(folder).



```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 Current directory changed.
```

Figure 72. Work with Object Menu with Current Directory Changed Message

Or, use the **CD** command to change current directory. Enter **cd mydir** on the command line.

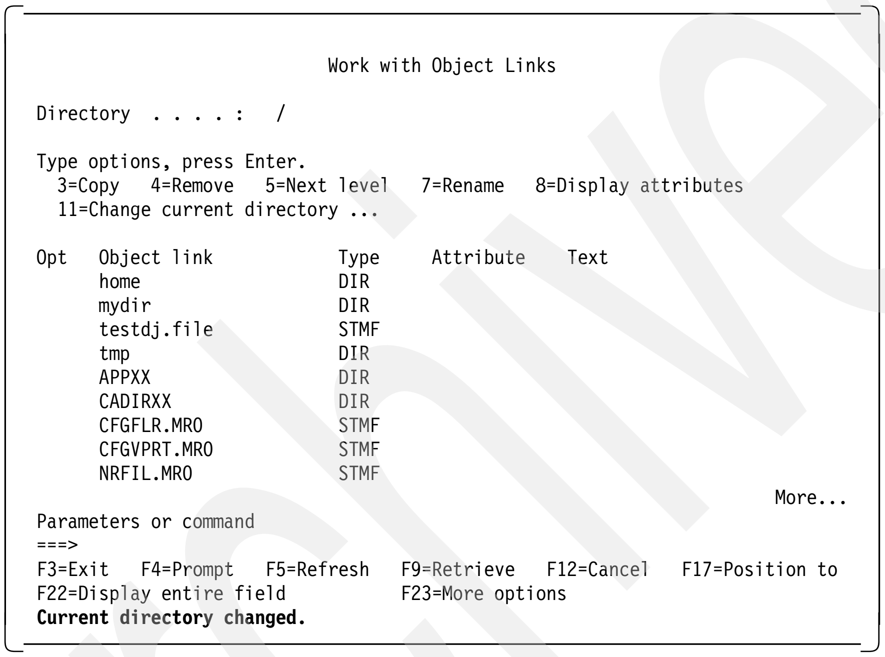


```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       testdj.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                     More...
 Parameters or command
 ===> cd mydir
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 73. Work with Object Menu with CD Command Specified

Press Enter; then the Current directory changed message is displayed.

The ″CHGCURDIR″ command does the same work.

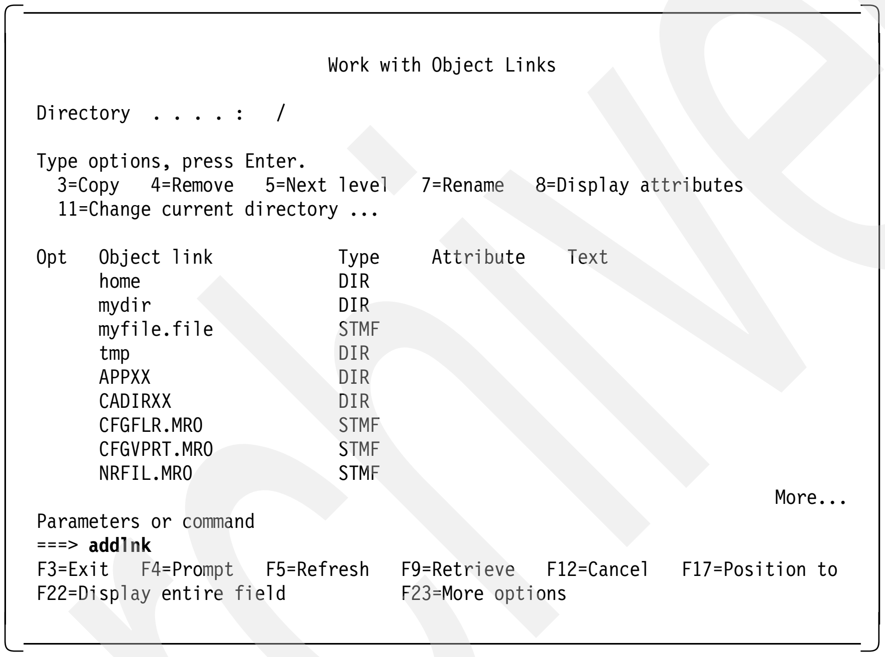


```
                             Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       testdj.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 Current directory changed.
```

Figure 74. Work with Object Menu with Current Directory Changed Message

A.5 Add, Display and Remove Object Links

Add Object Links: When you want to add a link between a directory and an object, enter ADDLNK on the command line.

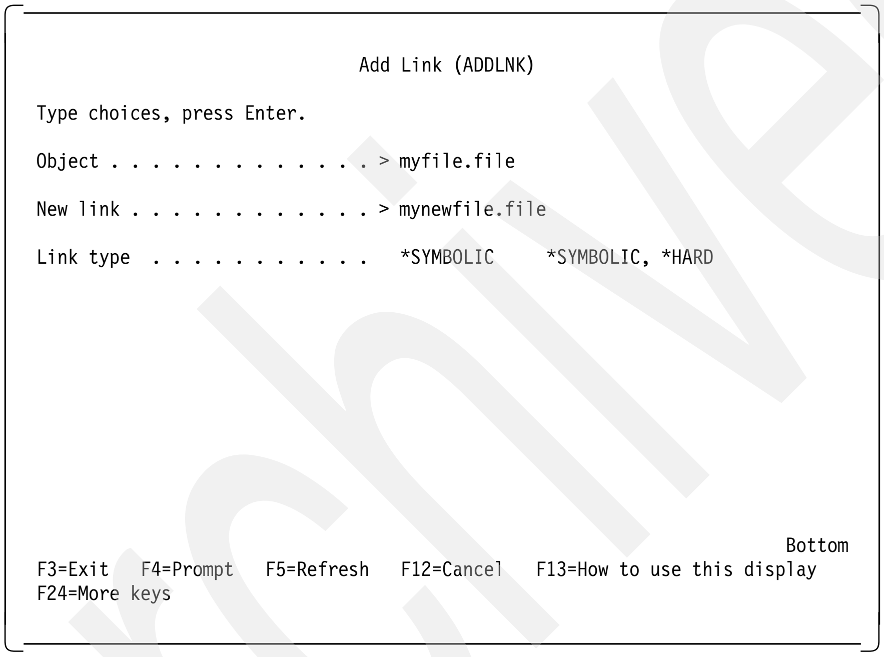


```
                             Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                       More...
 Parameters or command
 ===> addlnk
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 75. Work with Object Menu with ADDLNK Command Specified

Let′s assume the current directory is the ″root″ directory.

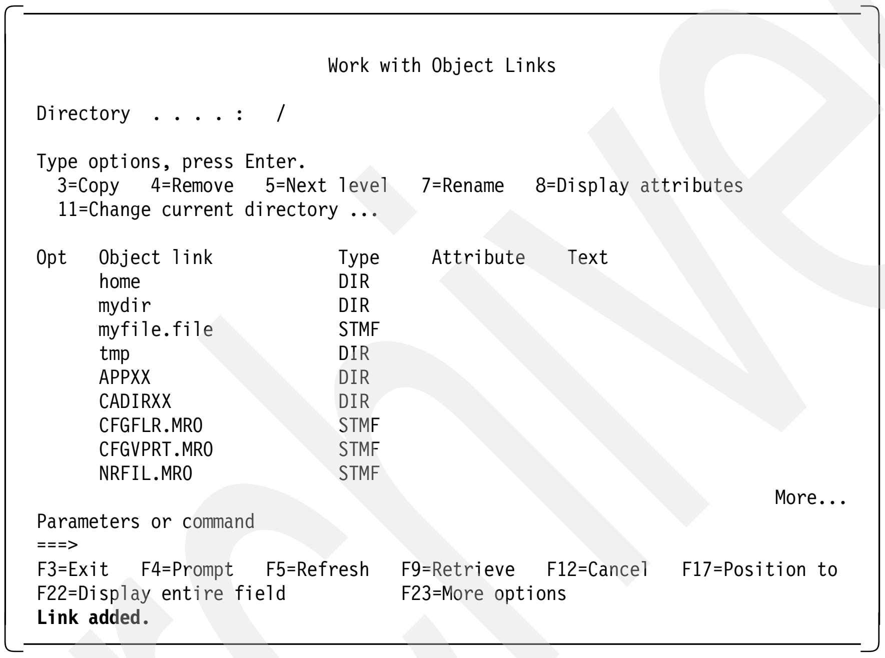
Enter ADDLNK on the command line and press F4; then the Add Link prompt text display is shown.



```
                              Add Link (ADDLNK)

 Type choices, press Enter.

 Object . . . . . . . . . . . . . > myfile.file

 New link . . . . . . . . . . . . > mynewfile.file

 Link type  . . . . . . . . . . .   *SYMBOLIC     *SYMBOLIC, *HARD









                                                                       Bottom
 F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
 F24=More keys
```

Figure 76. Add Link Prompt Text

Enter the input and press Enter; then the **Link added** message is displayed on the Work with Object Links display. You can get refreshed list by pressing the F5 key.



```
                              Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 Link added.
```

Figure 77. Work with Object Menu with Link Added Message

Since the syntax of each command in the integrated file system is somewhat different from the UNIX command, the command prompt (F4 key) is very useful to use command.

Display object links: When you want to see the object links, enter **DSPLNK** on the command line and press Enter.

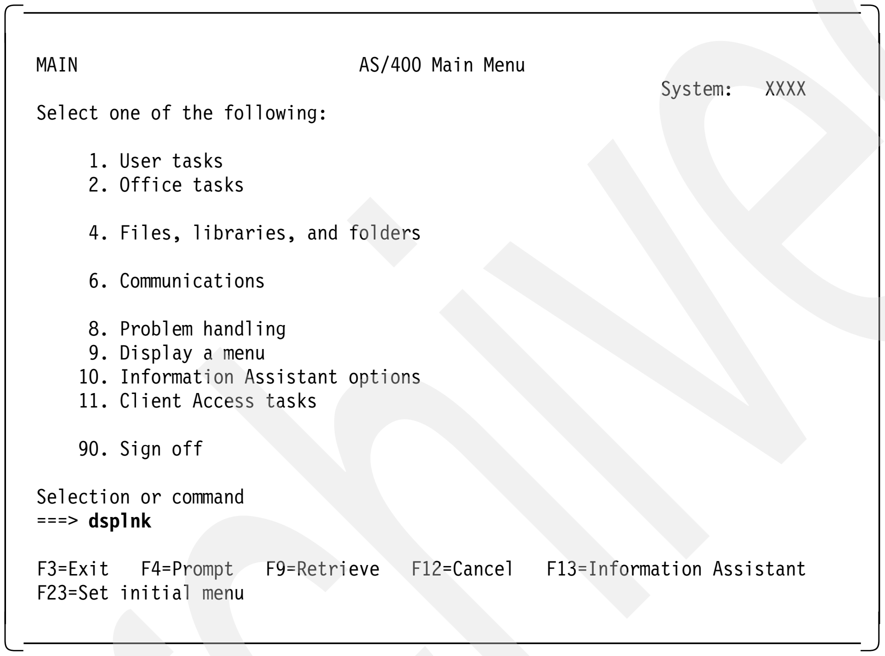
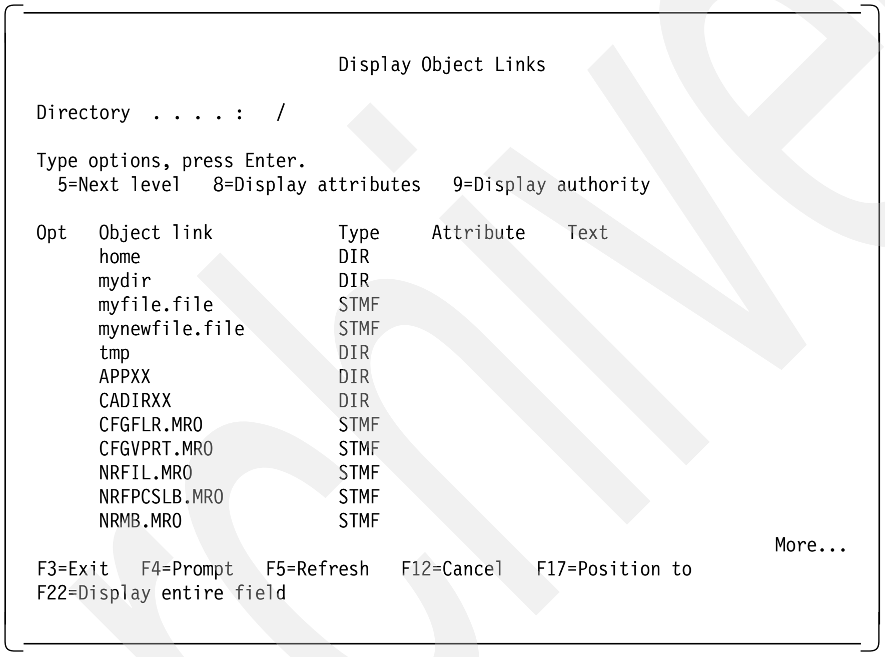

```
MAIN                           AS/400 Main Menu
                                                            System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> dsplnk

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
```

Figure 78. AS/400 Main Menu with DSPLNK Command Specified

Then the Display Object Links menu shows a list of objects in a current directory and provides options to display information about the objects.

In this case, the list for ″root″ directory is displayed because root directory is the current directory.

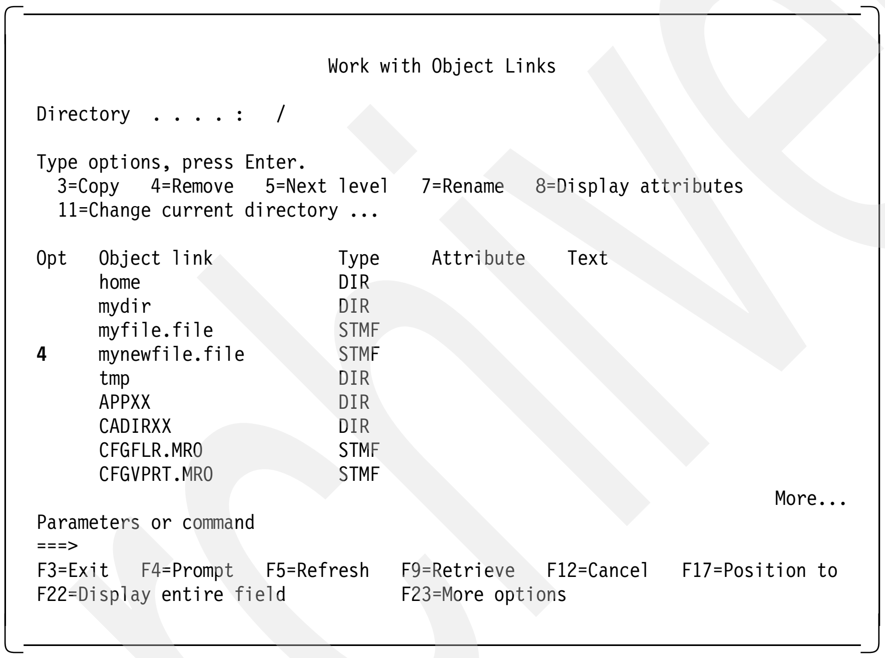


```
                             Display Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   5=Next level   8=Display attributes   9=Display authority

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       mynewfile.file         STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
       NRFPCSLB.MRO           STMF
       NRMB.MRO               STMF
                                                                       More...
 F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F17=Position to
 F22=Display entire field
```

Figure 79. Display Object Links Menu

Remove Object Links: To remove the link to an object, you can use option **4** or use the **del** and **ERASE** commands.

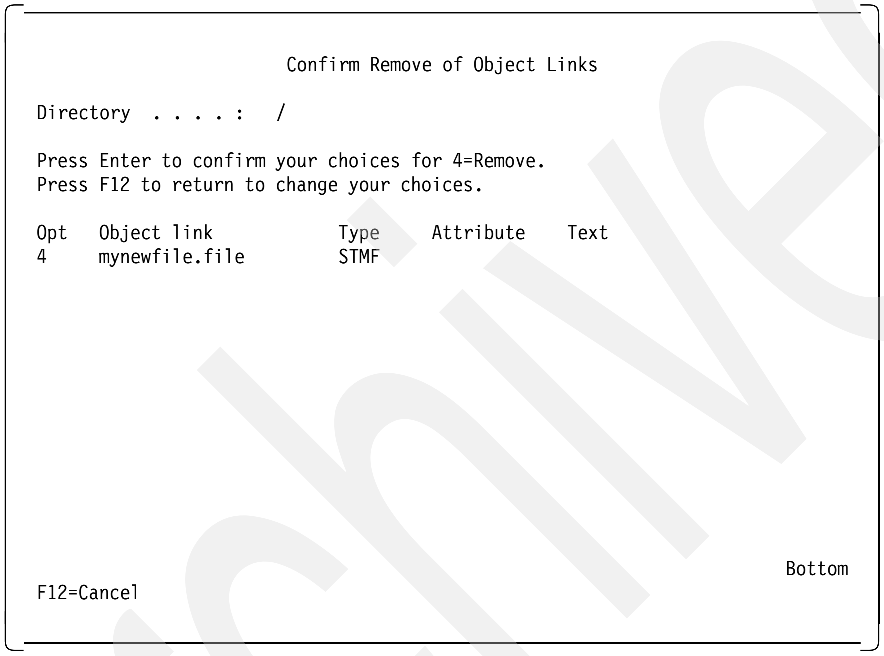
When you want to remove the link to an object, then enter **4** in the Opt column for the object or enter **del** on command line.



```
                             Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
 4     mynewfile.file         STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 80. Work with Object Menu with Option 4 Specified

Enter 4 in the Opt column for ″mynewfile.file″ and press Enter; then the Confirm Remove of Object Links display is shown.

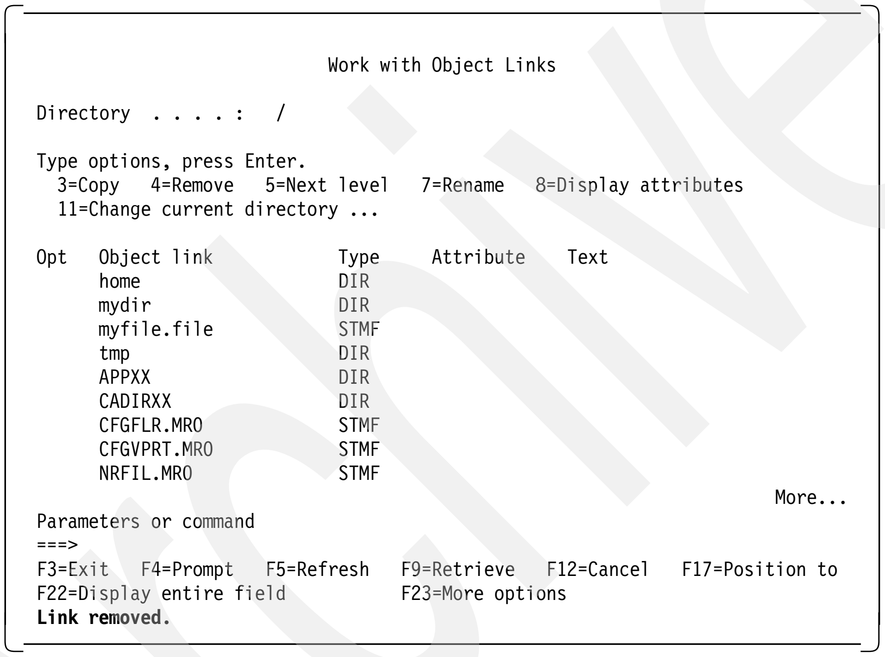


```
                        Confirm Remove of Object Links

 Directory  . . . . :   /

 Press Enter to confirm your choices for 4=Remove.
 Press F12 to return to change your choices.

 Opt   Object link           Type      Attribute    Text
 4     mynewfile.file        STMF












                                                                             Bottom
 F12=Cancel
```

Figure 81. Confirm Remove of Object Links Menu

And press Enter again; then ″mynewfile.file″ disappears and the **Link removed** message is displayed. You can get a refreshed list by pressing the F5 key.

Note: The object is also deleted if there are no other hard links to it and it is not in use.

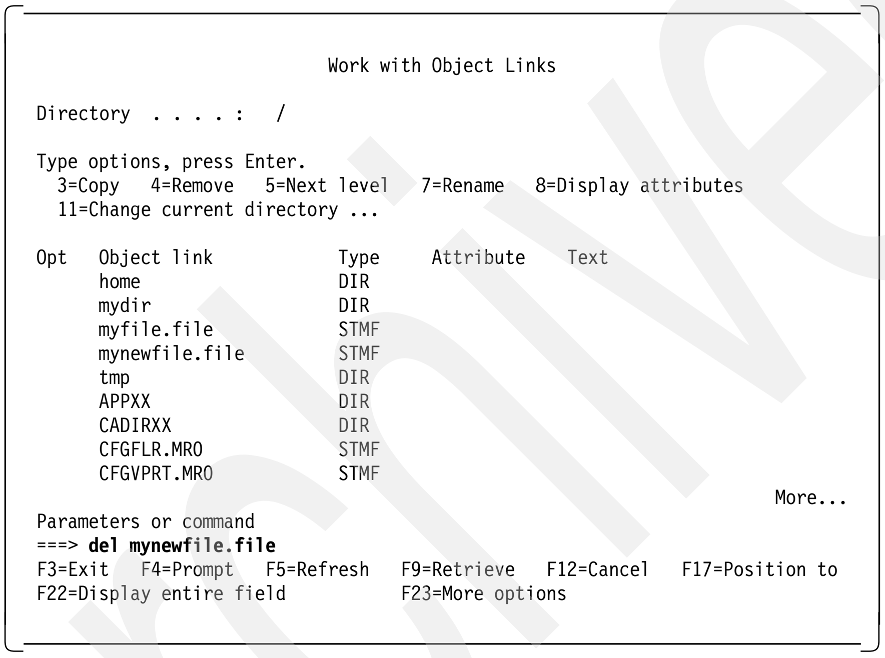


```
                           Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                     More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 Link removed.
```

Figure 82. Work with Object Menu with Link Removed Message

Let′s use the **del** command on the command line to remove the link of ″mynewfile.file″.

The **ERASE** command does the same work.



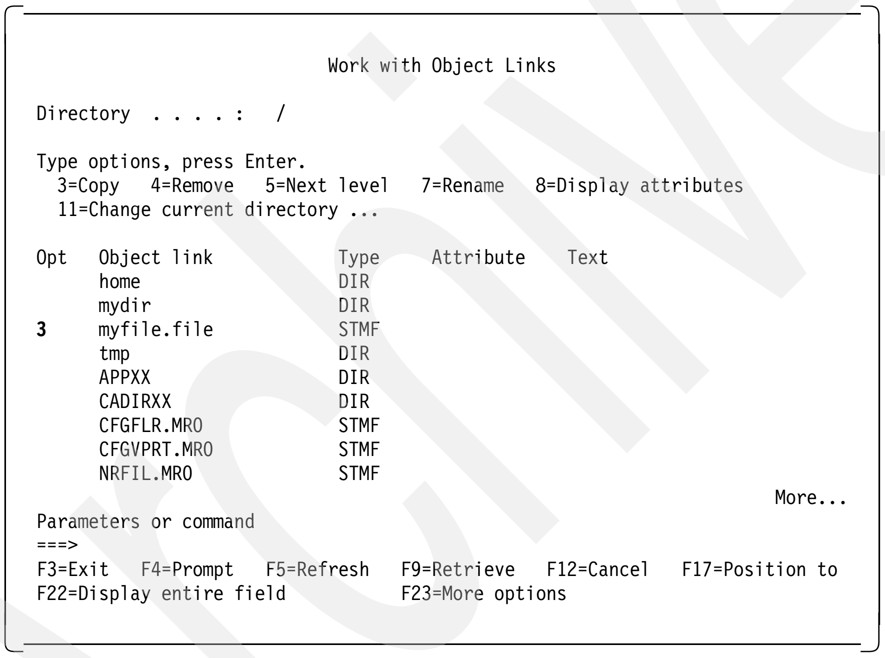
```
                          Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type     Attribute    Text
      home                   DIR
      mydir                  DIR
      myfile.file            STMF
      mynewfile.file         STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
                                                                       More...
Parameters or command
===> del mynewfile.file
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 83. Work with Object Menu with DEL Command Specified

A.6 Copy, Move, and Rename Objects

Copy Objects: To copy a single object or a group of objects, you can use option **3** or the **copy** and **CPY** commands.

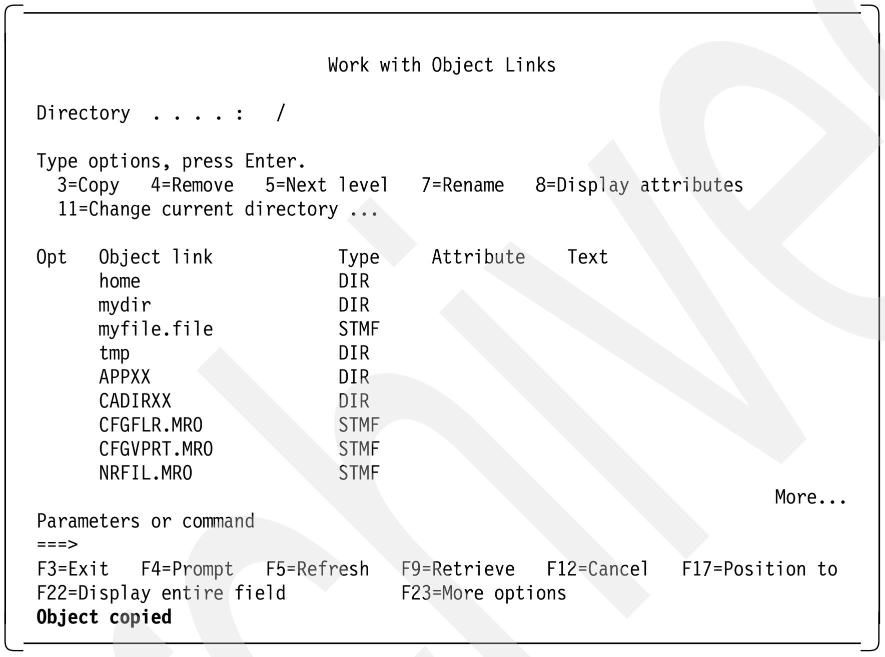
When you want to copy a single object or a group of objects, then enter **3** in the Opt column for the object or enter **copy** on the command line.



```
                          Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type      Attribute    Text
      home                   DIR
      mydir                  DIR
3     myfile.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
      NRFIL.MRO              STMF
                                                                   More...
Parameters or command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 84. Work with Object Menu with Option 3 Specified

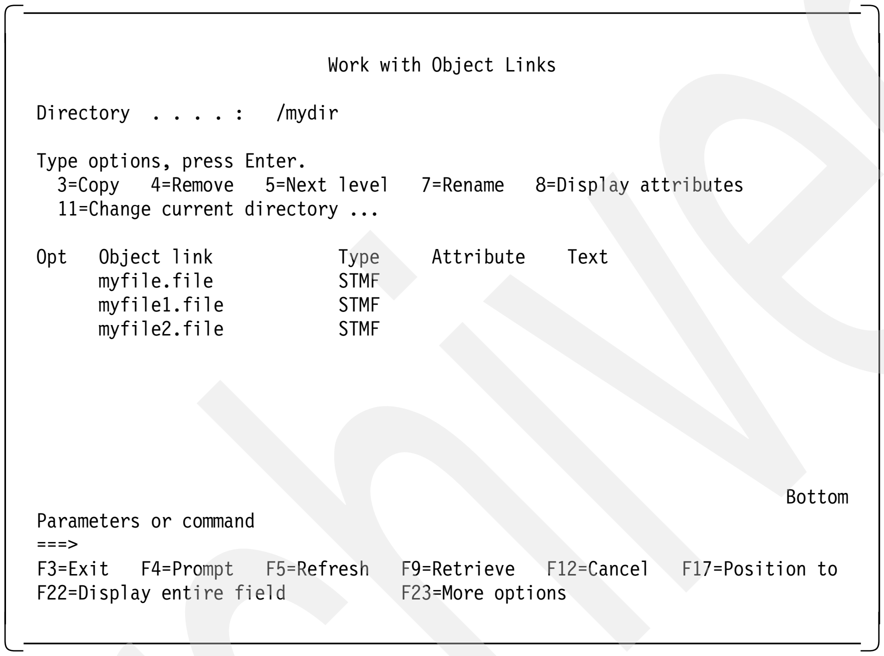
If current directory is ″mydir″ directory and you enter 3 on ″myfile.file″ and press Enter, then the Object copied message is displayed.



```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                      More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 Object copied
```

Figure 85. Work with Object Menu with Object Copied Message

And you can see ″myfile.file″ is copyed in ″mydir″ directory.

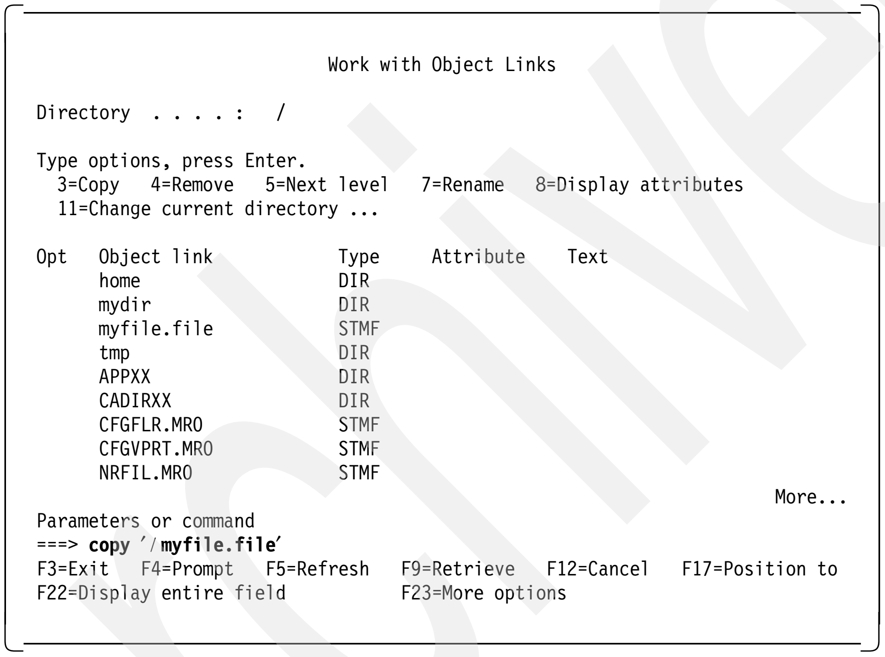


```
                            Work with Object Links

 Directory  . . . . :   /mydir

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type      Attribute    Text
       myfile.file            STMF
       myfile1.file           STMF
       myfile2.file           STMF




                                                                        Bottom
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 86. Work with Object Menu with Next Level

Let′s use the **copy** command on the command line to copy ″myfile.file″ to ″mydir″ directory. You can see ″myfile.file″ is copied in ″mydir″ directory also.

The **CPY** command does the same work.

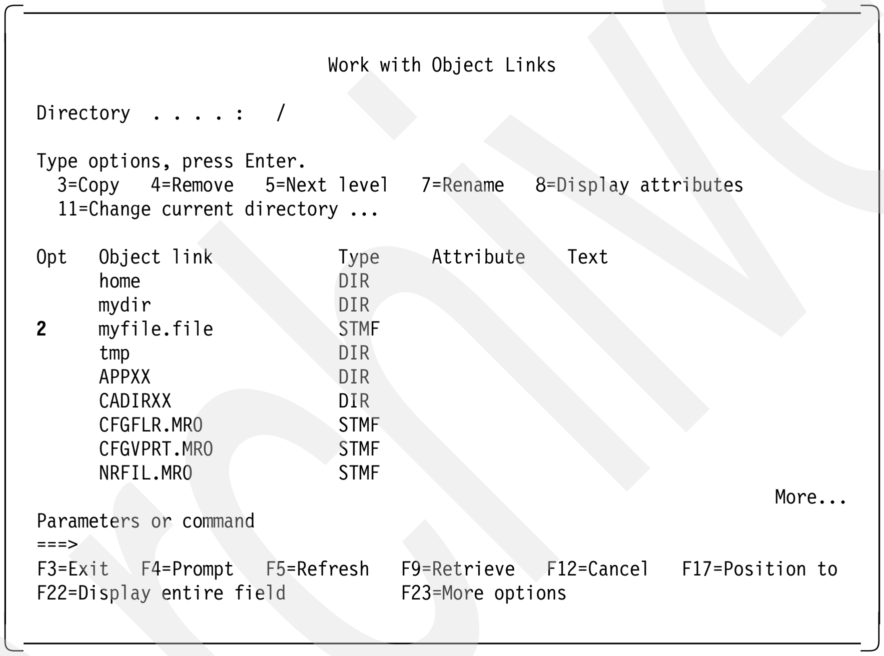


```
                             Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       myfile.file            STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
                                                                       More...
 Parameters or command
 ===> copy ′/myfile.file′
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 87. Work with Object Menu with Copy Command Specified

Move Objects: To move a single object or a group of objects, you can use option **2** or the **move** and **MOV** commands.

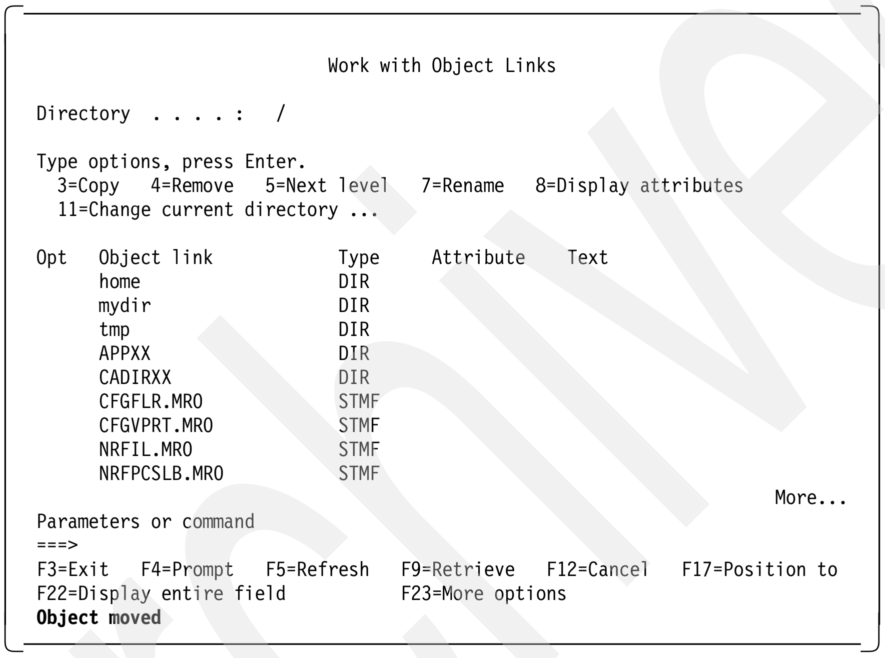
When you want to move a single object or a group of objects, then enter **2** in the Opt column for the object or enter **move** on the command line. The F23 key shows 2 option is move.



```
                            Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type     Attribute    Text
      home                   DIR
      mydir                  DIR
2     myfile.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
      NRFIL.MRO              STMF
                                                                   More...
Parameters or command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 88. Work with Object Menu with Option 2 Specified

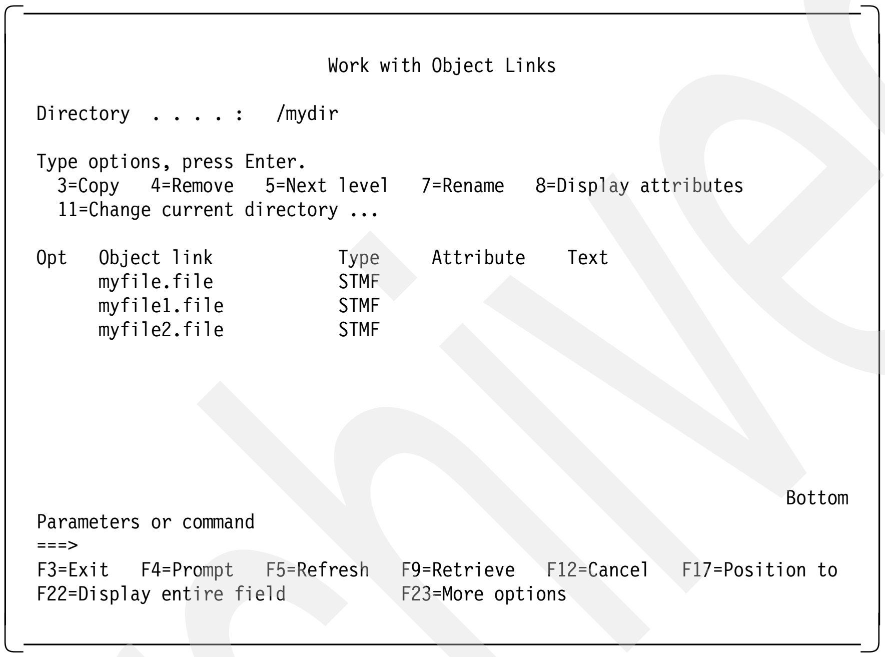
If the current directory is ″mydir″ directory and you enter 2 on ″myfile.file″ and press Enter, then ″myfile.file″ disappears and the **Object moved** message is displayed.



```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
       NRFPCSLB.MRO           STMF
                                                                         More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 Object moved
```

Figure 89. Work with Object Menu with Object Moved Message

You can see ″myfile.file″ is moved in ″mydir″ directory.

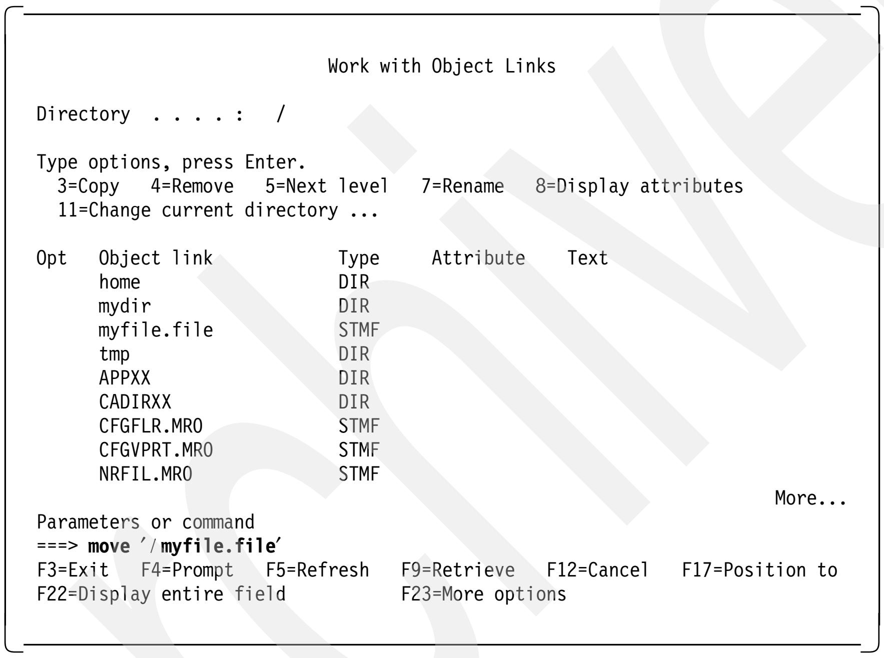


```
                             Work with Object Links

 Directory  . . . . :   /mydir

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type      Attribute    Text
       myfile.file            STMF
       myfile1.file           STMF
       myfile2.file           STMF







                                                                        Bottom
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 90. Work with Object Menu with Next Level

Let′s use the **move** command on the command line to move ″myfile.file″ to ″mydir″ directory. You can see ″myfile.file″ is moved in ′mydir′ directory also.

MOV command does the same work.

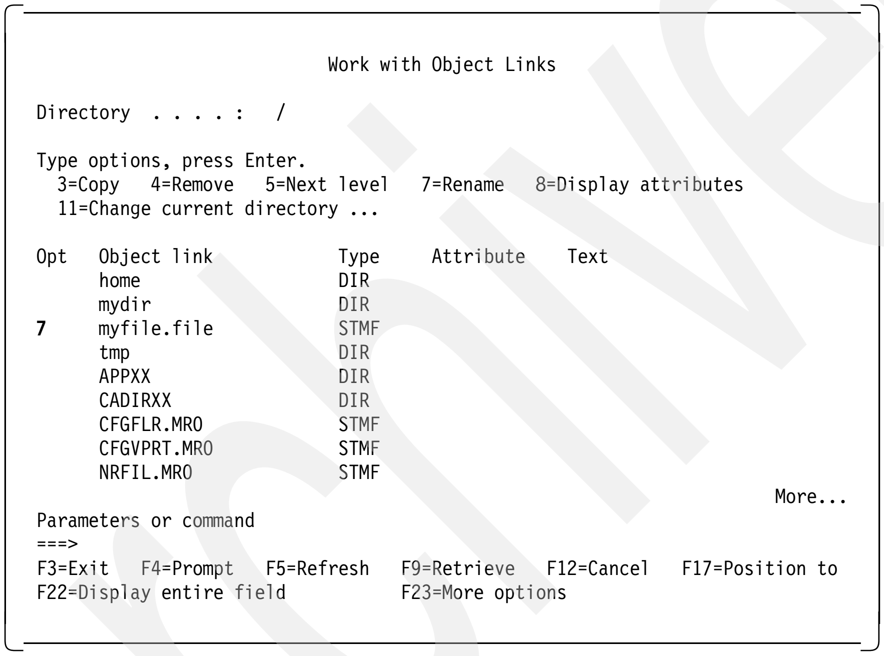


```
                            Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
  11=Change current directory ...

Opt   Object link            Type     Attribute    Text
      home                   DIR
      mydir                  DIR
      myfile.file            STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
      CFGFLR.MRO             STMF
      CFGVPRT.MRO            STMF
      NRFIL.MRO              STMF
                                                                    More...
Parameters or command
===> move ′/myfile.file′
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 91. Work with Object Menu with Move Command Specified

Rename Objects: To change the name of an object, you can use option **7** or the **ren** and **RNM** commands.

When you want to change the name of an object, then enter **7** in the Opt column for the object or enter **ren** on the command line.

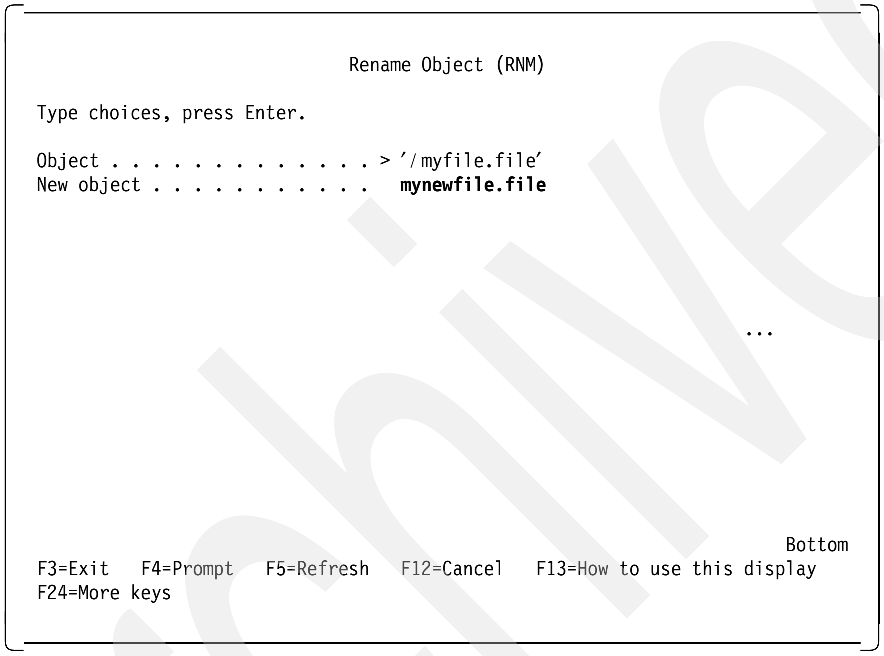


```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link           Type     Attribute    Text
       home                  DIR
       mydir                 DIR
 7     myfile.file           STMF
       tmp                   DIR
       APPXX                 DIR
       CADIRXX               DIR
       CFGFLR.MRO            STMF
       CFGVPRT.MRO           STMF
       NRFIL.MRO             STMF
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 92. Work with Object Menu with Option 7 Specified

Enter 7 on ″myfile.file″ and press Enter.

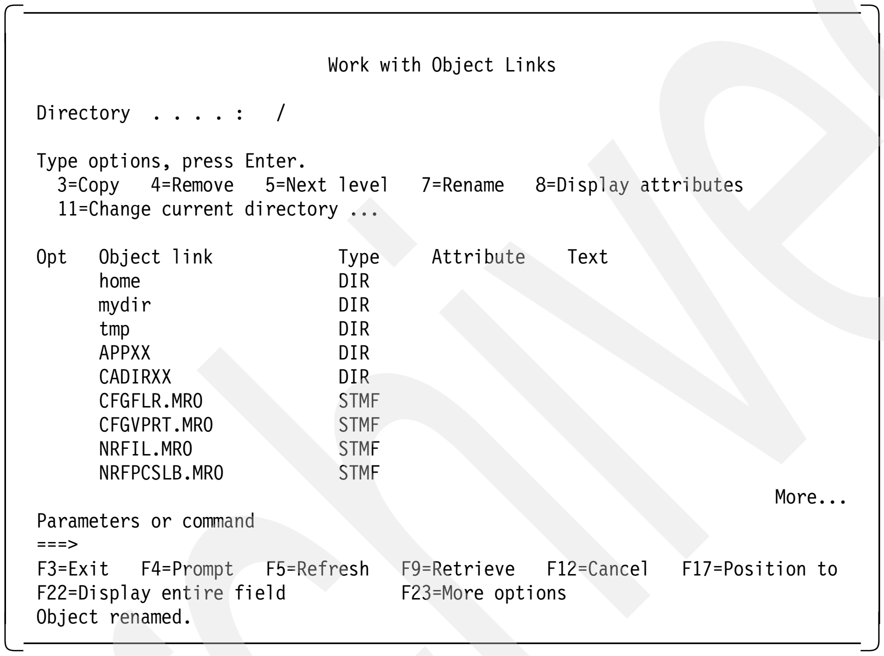
Then Rename Object display is shown and you can type a new name on the New Object field.



```
                        Rename Object (RNM)

Type choices, press Enter.

Object . . . . . . . . . . . . . > '/myfile.file'
New object . . . . . . . . . . .   mynewfile.file

                                                                      ...







                                                                          Bottom
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
```

Figure 93. Rename Object Prompt Text

Press Enter; then ′myfile.file′ disappears and the **Object renamed** message is displayed. You can get a refreshed list by pressing the F5 key.

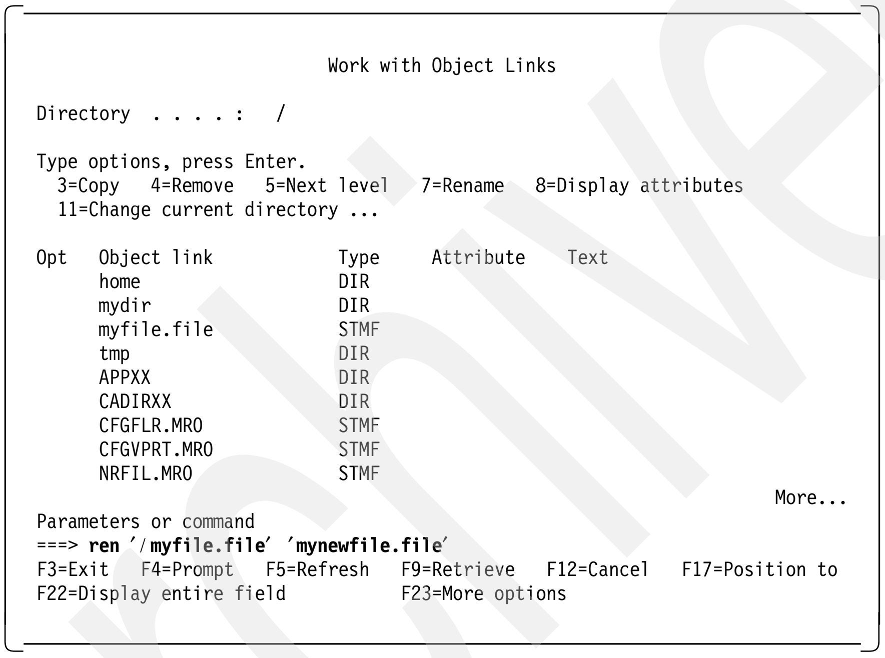


```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link            Type     Attribute    Text
       home                   DIR
       mydir                  DIR
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
       CFGFLR.MRO             STMF
       CFGVPRT.MRO            STMF
       NRFIL.MRO              STMF
       NRFPCSLB.MRO           STMF
                                                                      More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 Object renamed.
```

Figure 94. Work with Object Menu with Object Renamed Message

Let′s use the **ren** command on the command line to change ″myfile.file″ to another name.

The **RNM** command does the same work.



```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   3=Copy   4=Remove   5=Next level   7=Rename   8=Display attributes
   11=Change current directory ...

 Opt   Object link           Type     Attribute    Text
       home                  DIR
       mydir                 DIR
       myfile.file           STMF
       tmp                   DIR
       APPXX                 DIR
       CADIRXX               DIR
       CFGFLR.MRO            STMF
       CFGVPRT.MRO           STMF
       NRFIL.MRO             STMF
                                                                         More...
 Parameters or command
 ===> ren ′/myfile.file′ ′mynewfile.file′
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 95. Work with Object Menu with REN Command Specified

A.7 Other Tips

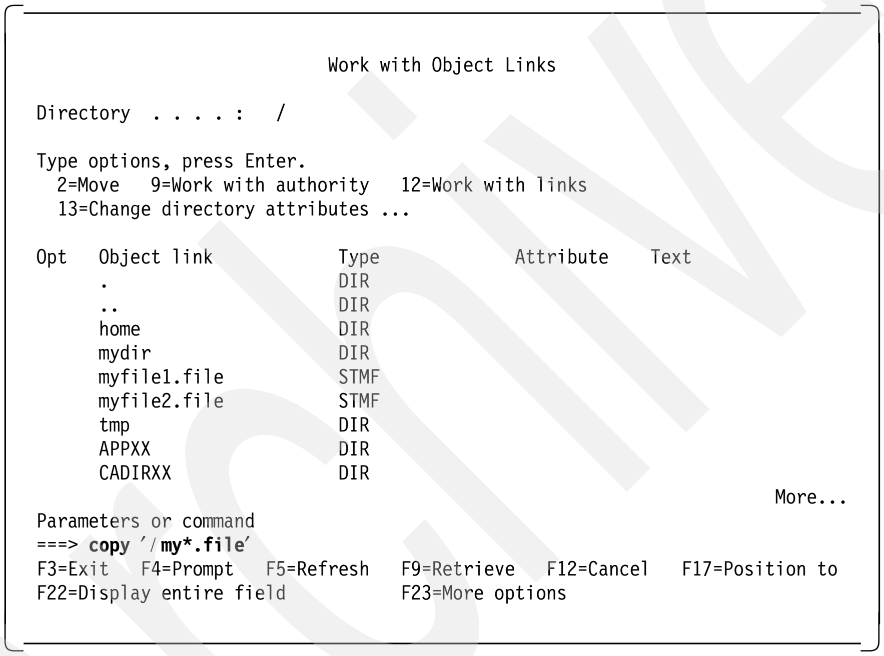
Wild Card Support: In some commands, an asterisk (*) or a question mark (?) can be used in the last component of a path name to search for patterns of names. The * tells the system to search for names that have any number of characters in the position of the * character. The ? tells the system to search for names that have a single character in the position of the ? character. The following example searches for all objects whose names begin with *d* and end with *txt*:

′/Dir1/Dir2/Dir3/d*txt′

The following example searches for objects whose names begin with *d* followed by any single character and end with *txt*:

′/Dir1/Dir2/Dir3/d?txt′

Let′s try to use the wild card with the COPY command. Let′s assume that the current directory is ″ydir″ and you want to copy two files starting with ″my″ in the ″root″ directory to ″mydir″ directory. Type `copy ′/my*.file′` on the command line and press Enter.



```
                            Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   2=Move   9=Work with authority   12=Work with links
   13=Change directory attributes ...

 Opt   Object link            Type              Attribute    Text
       .                      DIR
       ..                     DIR
       home                   DIR
       mydir                  DIR
       myfile1.file           STMF
       myfile2.file           STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
                                                                          More...
 Parameters or command
 ===> copy ′/my*.file′
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
```

Figure 96. Work with Object Menu with Copy Command Specified

Then you can get the success message if you have the authority for the two files.

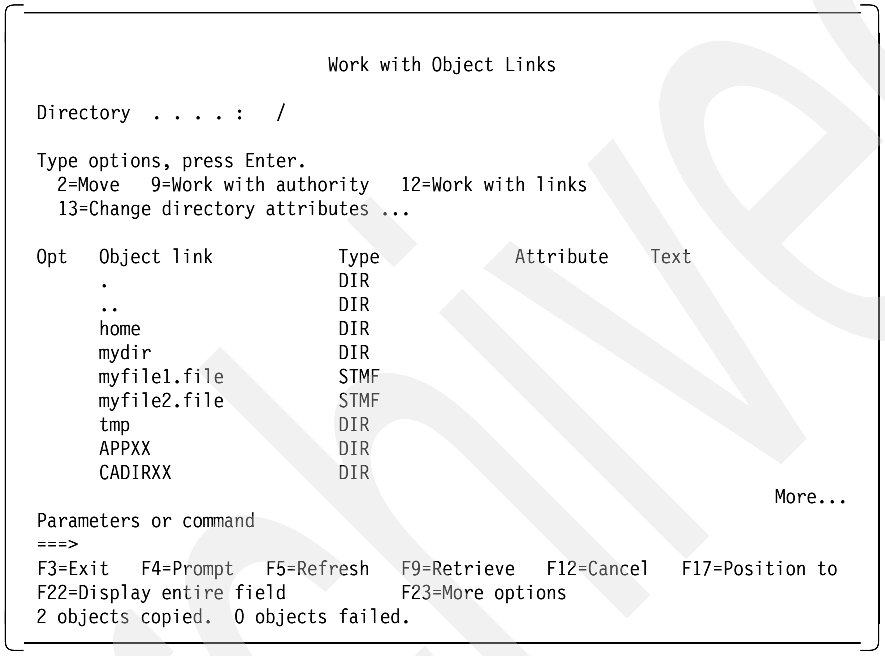
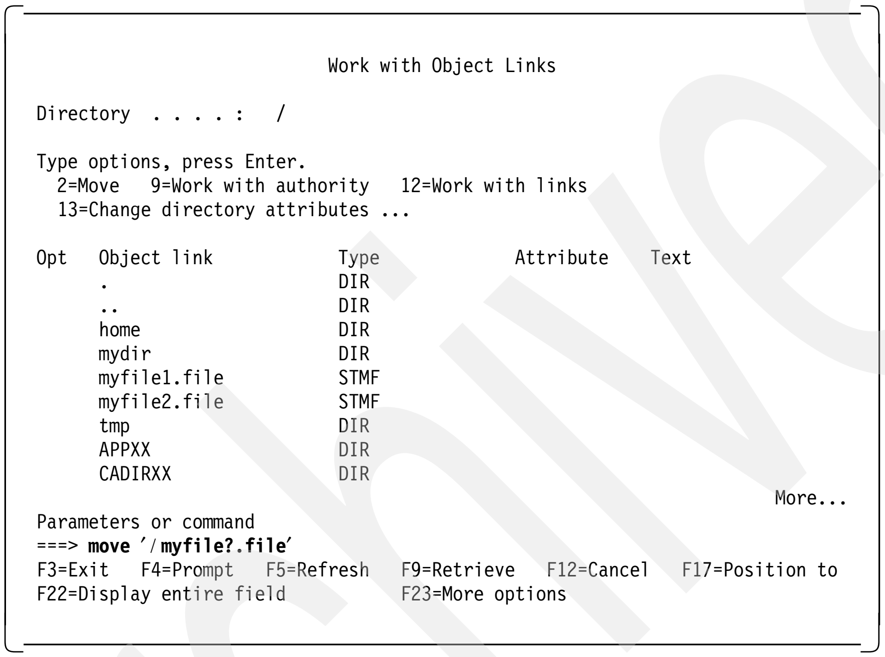

```
                          Work with Object Links

 Directory  . . . . :   /

 Type options, press Enter.
   2=Move   9=Work with authority   12=Work with links
   13=Change directory attributes ...

 Opt   Object link            Type             Attribute    Text
       .                      DIR
       ..                     DIR
       home                   DIR
       mydir                  DIR
       myfile1.file           STMF
       myfile2.file           STMF
       tmp                    DIR
       APPXX                  DIR
       CADIRXX                DIR
                                                                       More...
 Parameters or command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
 F22=Display entire field           F23=More options
 2 objects copied.  0 objects failed.
```

Figure 97. Work with Object Menu with Success Message

Or, test '?' with the MOVE command. '?' means a single wild character.



```
                            Work with Object Links

Directory  . . . . :   /

Type options, press Enter.
  2=Move   9=Work with authority   12=Work with links
  13=Change directory attributes ...

Opt   Object link            Type             Attribute    Text
      .                      DIR
      ..                     DIR
      home                   DIR
      mydir                  DIR
      myfile1.file           STMF
      myfile2.file           STMF
      tmp                    DIR
      APPXX                  DIR
      CADIRXX                DIR
                                                                          More...
Parameters or command
===> move '/myfile?.file'
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel   F17=Position to
F22=Display entire field           F23=More options
```

Figure 98. Work with Object Menu with Move Command Specified

Appendix B. Integrated File System Example Programs

We provide some example programs that might help you to understand the AS/400 integrated file system features better.

B.1 Client/Server Application for Stream File I/O

This example is a client/server application for maintaining customer lists. Server functions maintain a table for customer lists and can support the requests from multiple clients.

The server program contains five functions shown in the following list:

- Query
- Add
- Update
- Delete
- List

The client has the following display.

```
1. Query a Record
2. Add a Record
3. Update a Record
4. Delete a Record
5. Print all Record

6. Exit

Enter the number
```

Each client makes a request to a server and the server returns the results with a return code. Figure 99 on page 228 shows the flow chart briefly.

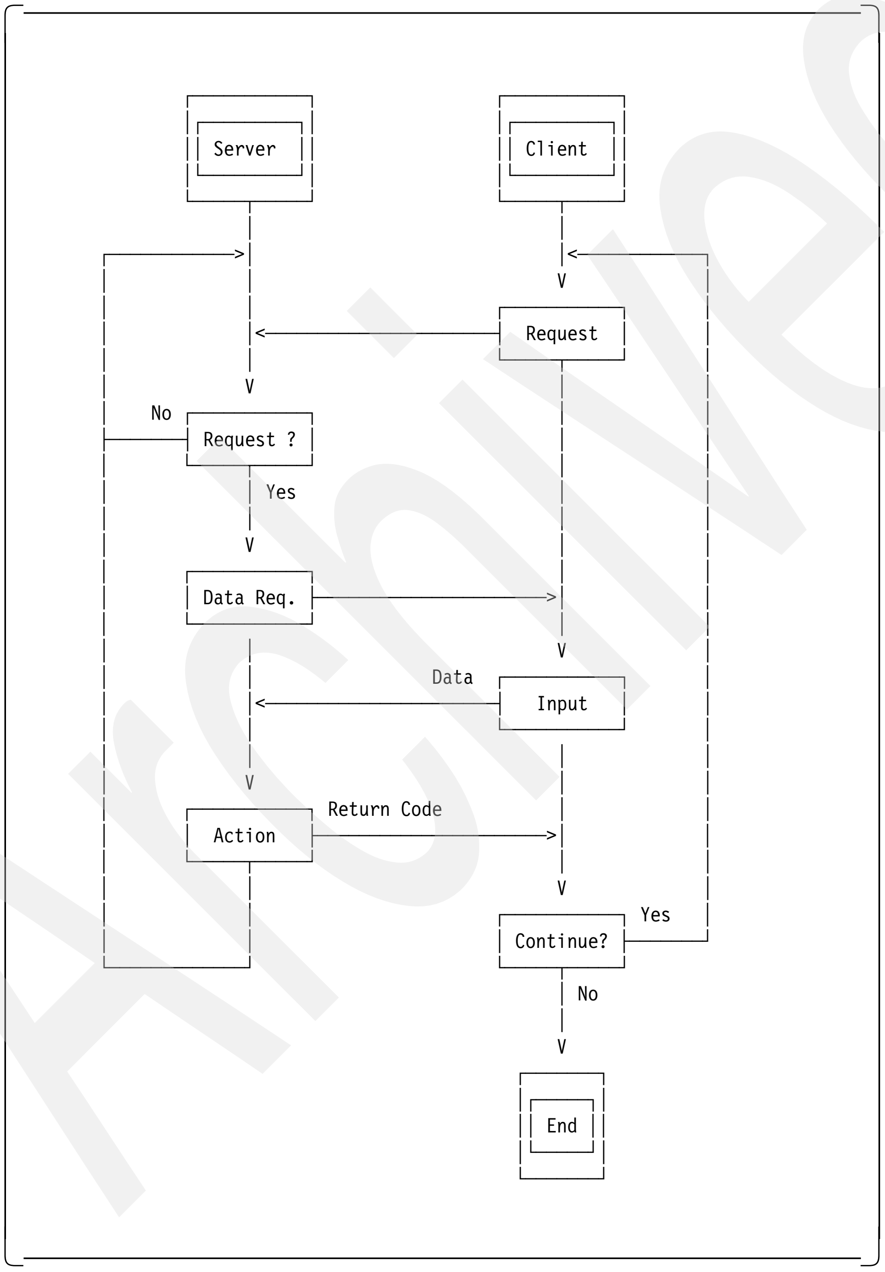


Figure 99. Flow Chart of Example 1 Program

The SERVER PGM consists of four MODULEs and The CLIENT PGM consists of two MODULEs. The module descriptions are as follows:

- SERVER PGM
  - SERVER MODULE - server main function.
  - DB MODULE - table maintenance function.
  - HASH MODULE - hash function.
  - MYRDWR MODULE - read, write function.
- CLIENT PGM
  - CLIENT MODULE - client main function.
  - MYRDWR MODULE - read, write function.

Marked lines with 1, 2, and 3 show what should be changed to run on the AS/400 system.

cs.h

```
#define BUFLEN  25
#define HOLDNUM 5
#define MAXCONN 5
#define QUIT    "QUIT"
#define UNDOMNM "/tmp/unixpipe"
#define NOFD    -1

#define EXIT_RC(a, b, c) {perror(a); close(b); return c;}


typedef enum rc
{
    RC_OK = 0, RC_SOCKET, RC_BIND,
    RC_LISTEN, RC_ACCEPT, RC_CLOSE,
    RC_EXEC, RC_READ, RC_WRITE,
    RC_CONNECT,
    RC_DBOPEN, RC_IDXOPEN
} rcode;

typedef struct fds
{
    int  sd;
    int  ld;
    int  ud;
```

```
} fds;



typedef enum _cmd
{
    DB_QRY = 0, DB_ADD, DB_UPD, DB_DEL, DB_ALL, DB_EXIT
} db_command;


typedef enum _key
{
    KEY_NO = 0, KEY_NAME
} db_key;


#define MAXNAMLEN  12
typedef struct _db {
    short    empno;
    char     name??(MAXNAMLEN??);
    short    age;
    char     sex;
    char     reserved;
} DBRECORD;

#define  NAMEOFFSET  (sizeof(short))

#define RECORDSIZE   sizeof(DBRECORD)

typedef struct _no_idx {
    short    empno;
    int      index;
    short    flag;
} NOINDEX;

typedef struct _name_idx {
    char     name??(MAXNAMLEN??);
    int      index;
    short    flag;
} NAMEINDEX;

#define   DELETED    1
typedef struct _hash {
    struct _hash  *next;
    char   *name;
    int    index;
    short  flag;
} HASH;

#define HASHSIZE  101

#define DBFILE    "/tmp/testdb"
#define DBIDXFILE   "/tmp/testidx"

typedef struct _msg_cli {
    db_command   cmd;
    db_key       key;
} CLIMSG;

typedef enum _ans {
    ANS_OK, ANS_EOF, ANS_READ, ANS_WRITE, ANS_UPDATE, ANS_DELETE,
    ANS_NOTFOUND, ANS_OTHER
} srv_rc;

/*
*  prototypes
*/

/* hash.c */
HASH   *lookup (char *name);
HASH   *insert (char *name, int index, short flag);

/* db.c */
rcode db_init(int *des);
srv_rc  db_rec_command (db_command cmd, DBRECORD *rec);
srv_rc db_set_all ();
srv_rc db_key_command (db_command cmd, char *name, DBRECORD *rec);

/* myrdwr.c */
#ifdef __ILEC400__    1
  int readn (int fd, void *ptr, int nbytes);
  int writen (int fd, void *ptr, int nbytes);
#else
  int readn (int fd, char *ptr, int nbytes);
  int writen (int fd, char *ptr, int nbytes);
#endif
```

Notes:

1 In this example, char pointers are used in UNIX and void pointers are used on the AS/400 system according to the difference in the default compiler option. Pointer type checking on the AS/400 system is stronger than UNIX.

server.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>      /* close, read, write    */
#include <sys/socket.h>  /* AF_INET, SOC_DGRAM... */
#include <netinet/in.h>  /* sockaddr_in           */
#include <string.h>      /* strstr()              */
#include <stdlib.h>      /* exit()                */

#ifdef __ILEC400__    2
  #include <time.h>
#else
  #include <sys/select.h>
#endif
#include <sys/un.h>

#include "cs.h"

static int high_val(int cnum, int sd, int *conn);
static rcode srv_ini(int *sd, char *buf);

static  db_command get_message(int fd, char *name, DBRECORD *rec)
{
    CLIMSG    msg;
    int       numbytes;
    srv_rc    rc;

    if ((numbytes = readn (fd, &msg, sizeof(CLIMSG))) != sizeof(CLIMSG))
        return -1;

    switch (msg.cmd) {
    case DB_ADD:
    case DB_UPD:
         if (readn (fd, rec, sizeof(DBRECORD)) != sizeof(DBRECORD))
             return -1;
         break;

    case DB_QRY:
    case DB_DEL:
        if (readn (fd, name, MAXNAMLEN) != MAXNAMLEN)
            return -1;
        break;
    }
    return msg.cmd;
}

static srv_rc  send_message(int fd, db_command cmd, char *name,
                                                    DBRECORD *rec)
{
    srv_rc  srvans;
    switch (cmd) {
    case DB_ADD:
    case DB_UPD:
        srvans = db_rec_command (cmd, rec);
        writen (fd, &srvans, sizeof (srv_rc));
        break;
    case DB_QRY:
        srvans = db_key_command (cmd, name, rec);
        writen (fd, &srvans, sizeof (srv_rc));
        if (srvans == ANS_OK)
            writen (fd, rec, sizeof (DBRECORD));
        break;
    case DB_DEL:
        srvans = db_key_command (cmd, name, rec);
        writen (fd, &srvans, sizeof (srv_rc));
        break;
    case DB_ALL:
  db_set_all();
        srvans = db_key_command (cmd, name, rec);
        if (srvans != ANS_OK)
            break;
        while (srvans != ANS_EOF) {
            if (*rec->name != '[') {
                writen (fd, &srvans, sizeof (srv_rc));
                writen (fd, rec, sizeof(DBRECORD));
            }
            srvans = db_key_command (cmd, name, rec);
        } /* endwhile */
        writen (fd, &srvans, sizeof (srv_rc));
        break;
    }

}
```

```
int main(int argc, char **argv)
{
    int rc, numbyte;
    int sd, ld;
    char buf??(1024??);
    int conn??(MAXCONN??);
    int cnum = 0;
    fd_set source, ready;
    int i, sel_event, fdp1;

    char name??(MAXNAMLEN+1??);
    DBRECORD  rec;
    db_command cmd;

    if ((rc = srv_ini(&ld, buf)) != RC_OK)
        return rc;

    FD_ZERO(&source);
    FD_SET(ld, &source);

    while(1) {
        fdp1 = high_val(cnum, ld, conn) + 1; /* Find the highest fd */

        /* Copy the contents of the master (source) fd_set
         * to the working (ready) fd_set.
         */
        memcpy((char *)&ready, (char *)&source, sizeof(ready));

        /* Wait for message */
        sel_event = select(fdp1,            /* Size of bit array */
                           &ready,          /* Sockets to listen to */
                           (void*) 0,       /* No Sockets writing */
                           (void*) 0,       /* No Error handling  */
                            NULL);          /* Time out period    */

        /*
         * sel_event == 0 for time out,  -1 for interrupt or Error
         * ready now has the socket(s) that have recv'd data
         */

        if (sel_event < 0) {
            continue;
        }

        /* If any new process has connected, register it. */
        if (FD_ISSET(ld, &ready)) {
            if (cnum == sizeof conn) {
                continue;
            }

            if ((sd = accept(ld, 0, 0)) < 0)
            EXIT_RC("Accept", sd, RC_ACCEPT)
            FD_SET(sd, &source);
            conn??(cnum++??) = sd;
        }

        for (i=0; i < cnum ; i++) {
            if (FD_ISSET(conn??(i??), &ready)) {
                if ((cmd = get_message(conn??(i??), name, &rec)) < 0) {
                    /* break the communication: Error or shutdown */
                    FD_CLR(conn??(i??), &source);
                    close(conn??(i??));

                    /* save_dbidx();  * index file will be saved */

                    /* move the last socket to the current place */
                    conn??(i??) = conn??(--cnum??);
                } else {
                    send_message(conn??(i??), cmd, name, &rec);
                }
            } /* if */
        } /* for */
    } /* while */

    return RC_OK;
}


 /*
  * high_val()
  * Calculate the highest used file descriptor in order
  * to use the correct mask in the select statement.
  */

static int high_val(int cnum, int sd, int *conn)
{
    int   hval, i;

    if (!cnum)
        return sd;
    else {
        for (hval = sd, i=0; i < cnum; i++)
            if (conn??(i??) > hval)
                hval = conn??(i??);
    }

    return hval;
}


 /*
 /* server_init()
 /* Initializes server socket and clears buffer
  */

static rcode srv_ini(int *des, char *buf)
{
    static struct sockaddr_un   server;
    int    sock_opt=1;

    unlink(UNDOMNM);
    if ((*des = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        EXIT_RC("Socket", *des, RC_SOCKET)

    setsockopt(*des,
               SOL_SOCKET,
               SO_REUSEADDR,
               (char *)&sock_opt,
               sizeof(sock_opt));

    memset(&server, '\0', sizeof(struct sockaddr_un));
    memset(buf, '\0', BUFLEN);

    server.sun_family      = AF_UNIX;
    strcpy (server.sun_path, UNDOMNM);

    if (bind(*des, (struct sockaddr *) &server,
             sizeof(struct sockaddr_un)) < 0)
        EXIT_RC("Bind", *des, RC_BIND)

    if (listen(*des, HOLDNUM) < 0)
        EXIT_RC("Listen", *des, RC_LISTEN)

    return db_init(des);
}
```

Notes:

2 <sys/select.h> header file is used in UNIX. It should be changed to <sys/types.h> and <time.h> header files for the AS/400 system.

myrdwr.c

```
#include <unistd.h>

int readn (int fd, void *vptr, int nbytes)     3
{
 char *ptr = vptr;                             3
 int nleft, nread;
 nleft = nbytes;

   while (nleft > 0)
   {
    nread = read(fd, ptr, nleft);
    if (nread < 0)
     return nread;
    else if (nread == 0)
     break;
    nleft -= nread;
    ptr += nread;
   }
 return (nbytes - nleft);
}

int writen (int fd, void *vptr, int nbytes)    3
{
 char *ptr = vptr;                             3
 int nleft, nwritten;
 nleft = nbytes;

   while (nleft > 0)
   {
    nwritten = write (fd, ptr, nleft) ;
    if (nwritten <= 0)
     return nwritten;
    nleft -= nwritten;
    ptr += nwritten;
   }
 return (nbytes - nleft);
}
```

Notes:

> **3** Char pointers are used in UNIX and void pointers are used on the AS/400 system according to the difference in the default compiler option. Pointer type checking on the AS/400 system is stronger than UNIX.

db.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cs.h"

#include <fcntl.h>
#include <errno.h>

static int  db_fd;
static int  idx_fd;
static int  serno;
static int  idxno;

rcode db_init(int *des)
{
    serno = 0;
    if ((db_fd = open(DBFILE, O_RDWR)) < 0) {
        if ((db_fd = creat(DBFILE, O_RDWR)) < 0)
            EXIT_RC("Open DB", *des, RC_DBOPEN);
    } else {
        if ((idx_fd = open(DBIDXFILE, O_RDWR)) < 0) {
            if ((idx_fd = creat(DBIDXFILE, O_RDWR)) < 0)
                EXIT_RC ("Open IDX", *des, RC_IDXOPEN);
        } else {
            NAMEINDEX  nameidx;
            while (read(idx_fd, &nameidx, sizeof(nameidx)) > 0)
                insert (nameidx.name, nameidx.index, nameidx.flag );
        }
    }
    return RC_OK;
}
```

```
srv_rc  db_rec_command (db_command cmd, DBRECORD *rec)
{
    HASH  *hp;
    NAMEINDEX nmidx;

    switch (cmd) {
    case DB_ADD:
        lseek (db_fd, serno*sizeof(DBRECORD), SEEK_SET);
        if (errno < 0)
            return ANS_OTHER;
        if (write (db_fd, rec, sizeof(DBRECORD)) != sizeof(DBRECORD))
            return ANS_WRITE;
        hp = insert (rec->name, serno++, 0);
        lseek (idx_fd, idxno*sizeof(DBRECORD), SEEK_SET);
        idxno++;
        memcpy (nmidx.name, hp->name, MAXNAMLEN);
        nmidx.index = hp->index;
        nmidx.flag = hp->flag;
        write (idx_fd, &nmidx, sizeof (NAMEINDEX));
        break;
    case DB_UPD:
        hp = lookup (rec->name);
        if (hp == NULL)
            return ANS_NOTFOUND;
        lseek (db_fd, hp->index*sizeof(DBRECORD), SEEK_SET);
        if (errno < 0)
            return ANS_OTHER;
        if (write (db_fd, rec, sizeof(DBRECORD)) != sizeof(DBRECORD))
            return ANS_UPDATE;
    }
    return ANS_OK;
}

srv_rc db_set_all ()
{
    lseek (db_fd, 0, SEEK_SET);
    if (errno < 0)
        return ANS_OTHER;
    return ANS_OK;
}


srv_rc db_key_command (db_command cmd, char *name, DBRECORD *rec)
{
    HASH    *hp;
    int  numbyte;

    switch (cmd) {
    case DB_QRY:
    case DB_DEL:
        hp = lookup (name);
        if (hp == NULL)
            return ANS_NOTFOUND;

        lseek (db_fd, hp->index*sizeof(DBRECORD), SEEK_SET);
        if (errno < 0)
            return ANS_OTHER;
        if (cmd == DB_QRY) {
            if (read (db_fd, rec, sizeof(DBRECORD)) != sizeof(DBRECORD))
                return ANS_READ;
        } else if (cmd == DB_DEL) {
            NAMEINDEX nmidx;
            memcpy (nmidx.name, name, MAXNAMLEN);
            nmidx.flag = hp->flag = DELETED;
            nmidx.name??(0??) = hp->name??(0??) = '[';
            nmidx.index = hp->index;
            lseek(idx_fd, hp->index*sizeof(NAMEINDEX), SEEK_SET);
            write (idx_fd, &nmidx, sizeof(NAMEINDEX));
            lseek(db_fd, NAMEOFFSET, SEEK_CUR);
            if (write (db_fd, nmidx.name, 1) != 1)
                return ANS_DELETE;
        }
        break;
    case DB_ALL:
        if ((numbyte = read (db_fd, rec, sizeof(DBRECORD))) == 0)
            return ANS_EOF;
        else if (numbyte != sizeof(DBRECORD))
            return ANS_READ;
        break;
    }
    return ANS_OK;
}
```

hash.c

```
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>

#include ″cs.h″

static HASH    *hashtab??(HASHSIZE??);

static unsigned  hash(char *str)
{
    unsigned  hashval;
    int       i;
    for (hashval = i = 0; i < MAXNAMLEN ; i++, str++) {
        hashval = *str + 31 * hashval;
    }
    return hashval % HASHSIZE;
}

HASH    *lookup (char *name)
{
    HASH  *hp;

    for (hp = hashtab??(hash(name)??) ; hp ; hp = hp->next) {
        if (memcmp(name, hp->name, MAXNAMLEN) == 0 && hp->flag != DELETED)
            return hp;
    }
    return hp;
}

HASH   *insert (char *name, int index, short flag)
{
    HASH  *hp;
    unsigned hashval;

    if ((hp = lookup(name)) == NULL) {
        hp = malloc (sizeof(HASH));
        if (hp == NULL || (hp->name = malloc(MAXNAMLEN)) == NULL )
            return NULL;
        memcpy (hp->name, name, MAXNAMLEN);
        hashval = hash(name);
        hp->next = hashtab??(hashval??);
        hashtab??(hashval??) = hp;
    }
    hp->index = index;
    hp->flag = flag;
    return hp;
}
```

client.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>      /* close, read, write    */
#include <sys/socket.h>  /* AF_INET, SOC_DGRAM... */
#include <string.h>      /* strstr()              */
#include <stdlib.h>      /* exit()                */

#include <sys/un.h>

#include "cs.h"

static rcode cleanup(fds *desc, char *msg, rcode ret);

static rcode cleanup(fds *desc, char *msg, rcode ret)
{
    perror(msg);

    if (desc->ld > NOFD)
        close(desc->ld);
    if (desc->ud > NOFD)
        close(desc->ud);
    if (desc->sd > NOFD)
        close(desc->sd);
    return ret;
}


static db_command  get_command()
{
     char   buf??(128??);
     db_command  cmd??(6??) = {DB_QRY, DB_ADD, DB_UPD, DB_DEL, DB_ALL, DB_EXIT};

     int    opt;
     while (1) {
         printf ("\n\n");
         printf ("1. Query a Record\n");
         printf ("2. Add a Record\n");
         printf ("3. Update a Record\n");
         printf ("4. Delete a Record\n");
         printf ("5. Print all Record\n\n");
         printf ("6. Exit\n\n");
         printf ("Enter the number\n");
         gets (buf);

         if ((opt = atoi(buf)) > 0 && opt < 7)
             return cmd??(opt-1??);
         else
             printf ("Invalid Option!\n");
     }
}

static void get_str(char *msg, int maxlen, char *name)
{
    char buff??(1024??);
    printf (msg);
    gets(buff);
    if (strlen(buff) < maxlen)
        strcat(buff, "                                        ");
    memcpy (name, buff, maxlen);
}

static void get_updstr(char *msg, int maxlen, char *name)
{
    char buff??(1024??);
    int  len;
    printf (msg);
    gets(buff);
    if ((len=strlen(buff)) == 0)
        return;
    if (strlen(buff) < maxlen)
        strcat(buff, "                                                  ");
    memcpy (name, buff, maxlen);
}

static void get_int (char *msg, int *ibuf)
{
    char buff??(1024??);
    printf(msg);
    voidgets(buff);
    *ibuf = atoi(buff);
}
```

```
static void get_updint (char *msg, int *ibuf)
{
    char buff??(1024??);
    printf(msg);
    gets(buff);
    if (strlen(buff) == 0)
        return;
    *ibuf = atoi(buff);
}

static void get_record (DBRECORD *rec)
{
    int   age;
    get_str("Enter name\n", MAXNAMLEN, rec->name);
    get_str("Enter gender(F/M)\n", 1, &rec->sex);
    get_int("Enter age\n", &age);
    rec->age = age;
}

static void get_update (DBRECORD *rec)
{
    int  age = rec->age;
    get_updstr("Enter gender(F/M)\n", 1, &rec->sex);
    get_updint("Enter age\n", &age);
    rec->age = age;
}

static void print_rec( DBRECORD *rec)
{
    char name??(MAXNAMLEN+1??);
    memcpy (name, rec->name, MAXNAMLEN);
    name??(MAXNAMLEN??) = '\0';
    printf ("Name: %s  Jender: %c  Age: %d\n", name, rec->sex, rec->age);
}

static void print_msg (srv_rc srvans)
{
    switch (srvans) {
    case ANS_OK:
         printf("OK\n");
         break;
    case ANS_READ:
         printf("READ error\n");
         break;
    case ANS_WRITE:
         printf("WRITE error\n");
         break;
    case ANS_UPDATE:
         printf("UPDATE error\n");
         break;
    case ANS_DELETE:
         printf("DELETE error\n");
         break;
    case ANS_NOTFOUND:
         printf("Record Notfound\n");
         break;
    case ANS_OTHER:
         printf("Other error\n");
         break;
    } /* endswitch */
}

int  main(int argc, char **argv)
{
    static struct sockaddr_un client;
    fds    desc;
    int    opt;
    int    numbytes;

    char buf??(1024??);
    char name??(MAXNAMLEN+1??);
    DBRECORD  rec;
    CLIMSG    msg, upd2;
    srv_rc    srvans;

    memset (&client, '\0', sizeof(struct sockaddr_un));
    client.sun_family      = AF_UNIX;
    strcpy (client.sun_path, UNDOMNM);

    if ((desc.ud = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        cleanup (&desc, "UNIX Socket", RC_SOCKET);
        exit(-1);
    }

    if (connect(desc.ud, (struct sockaddr *)&client,
                (int)SUN_LEN(&client)) < 0) {
        cleanup (&desc, "UNIX Connect", RC_CONNECT);
        exit(-1);
    }
```

```
while (1) {
    switch ((msg.cmd = get_command())) {
    case DB_EXIT:
        cleanup(&desc, "", RC_OK);
        exit(0);
    case DB_ADD:
        get_record(&rec);
         break;
    case DB_QRY:
    case DB_DEL:
    case DB_UPD:
         get_str("Enter name\n", MAXNAMLEN, name);
         break;
    case DB_ALL:
         break;
    }
    if (msg.cmd == DB_UPD) {
       memcpy (&upd2, &msg, sizeof(CLIMSG));
       msg.cmd = DB_QRY;
    }

    numbytes = writen (desc.ud, &msg, sizeof(CLIMSG));

    switch (msg.cmd) {
    case DB_ADD:
         numbytes = writen (desc.ud, &rec, sizeof(DBRECORD));
         break;
    case DB_QRY:
    case DB_DEL:
    case DB_UPD:
         writen (desc.ud, name, MAXNAMLEN);
         break;
    }

    readn (desc.ud, &srvans, sizeof (srv_rc));
    print_msg(srvans);

    if (upd2.cmd == DB_UPD) {
        msg.cmd = DB_UPD;
        upd2.cmd = -1;
    }

    switch (msg.cmd) {
    case DB_UPD:
```

```
                if (srvans == ANS_OK) {
                    readn (desc.ud, &rec, sizeof (DBRECORD));
                    get_update (&rec);
                    numbytes = writen (desc.ud, &msg, sizeof(CLIMSG));
                    numbytes = writen (desc.ud, &rec, sizeof(DBRECORD));
                    readn (desc.ud, &srvans, sizeof (srv_rc));
                    print_msg(srvans);
                }
                break;

           case DB_QRY:
                if (srvans == ANS_OK) {
                    readn (desc.ud, &rec, sizeof (DBRECORD));
                    print_rec(&rec);
                }
                break;
           case DB_ALL:
                if (srvans != ANS_OK)
                    break;
                while (srvans != ANS_EOF) {
                    readn (desc.ud, &rec, sizeof(DBRECORD));
                    print_rec(&rec);
                    readn (desc.ud, &srvans, sizeof (srv_rc));
                } /* endwhile */
                break;
           }
       }
       return 0;
   }
```

B.2 Display Stream File

There is no command to display the contents of the stream files on the AS/400 dummy display. This example shows the stream file on the display. It tests the argument path, if it is not a full path name, it is expanded to a full path name with the current working directory using the function getcwd(). This program opens a file, reads data from the file, and writes it on the display. As mentioned, the file descriptor 1 is not a standard file descriptor on the UNIX system. To display the data, the ILE C function fwrite() is used.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

#define  BUFFER_SIZE    4096
#define  FILENAME_SIZE  1024
#define  SEPERATOR      ″/″
#define  USAGE          ″DSPSTMF filename\n″


int      display_stmf(char *path)
{
    int      fd = -1;
    int      BytesRead;
    char     buffer[BUFFER_SIZE];
    char     filename[FILENAME_SIZE];
    .
    .

    strcpy (filename, path);
    if (!isfulldirectory(path)) {
        char   cwd[FILENAME_SIZE];
        int    iserror;
        iserror = getcwd(cwd, FILENAME_SIZE);
        ...
        makefullpath(filename,cwd);
    }

    if ((fd = open(filename, O_RDONLY, S_IRWXU)) == -1) {
        perror (″open Error″);
        exit (-1);
    }

    while ((BytesRead = read (fd, buffer, BUFFER_SIZE)) > 0) {
#ifdef __ILEC400__
        fwrite (buffer, 1, BytesRead, stdout);       1
#else
        write (1, buffer, BytesRead);                2
#endif
    }

    close(fd);
    return 0;
}
```

Note:

> **1** ILE C/400 simulates the standard output.
>
> **2** Write data on the file descriptor 1 does not work in this case.

B.3 Listing Directory

This example lists the files in a directory that are passed as parameters by the user. It takes the filename and the other information by calling the function stat(). If the file is a regular file, it prints the size of the file and its name. If the file is a directory, this example prints recursively.

This example shows what should be changed in the source code; the marked lines with **2**, **3**, and **4** are required if it runs not only on the the AS/400 system but also on the UNIX system.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <errno.h>

#define  MAX_PATH    1024

static void    fsize (char *);
static void    dirwalk (char *, void (*fcn)(char *));

int     main (int argc, char **argv)
{
    if (argc == 1)
        fsize(″.″);
    else
        while (--argc > 0)
            fsize (*++argv);
    return 0;
}

/* fsize:
   This function prints the size of the file.
   If the file is a directory, fsize first calls the function
   dirwark().
*/
static void    fsize(char *name)
{
    struct stat stbuf;
    if (stat(name, &stbuf) == -1) {       1
        perror("stat");
        return;
    }

#ifdef __ILEC400__                           2
    if (S_ISDIR(stbuf.st_mode))                      3
#else
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)     4
#endif
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}


/* dirwalk:
   The function dirwalk() opens the directory and read the entry
   of the directory. This function is called inderect recursively
   to get the files of the directory.
*/
static void    dirwalk (char *dir, void(*fcn)(char *))
{
    char name[MAX_PATH];
    struct dirent    *dp;
    DIR      *dfd;

    if ((dfd = opendir(dir)) == NULL) {       5
        perror ("opendir");
        return;
    }

    while (( dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->d_name, ".") == 0 ||
        strcmp(dp->d_name, "..") == 0)
            continue;
        sprintf(name, "%s/%s", dir, dp->d_name);
        (*fcn)(name);
    }
    closedir(dfd);
}
```

Notes:

1 This function takes the pointer to the filename and the structure of struct stat, which is defined in the header file <sys/stat.h>. The return value is either 0 or 1.

2 **3** Because the macros S_IFMT and S_IFDIR are not defined in the ILE C/400, the preprocessor directive #ifdef __ILEC400__ is required and should be used for the macro S_ISDIR(parm).

5 This function takes the pointer to the directory name and returns a NULL pointer if it fails. Otherwise, it returns the pointer to the structure of DIR, which is defined in the header file <dirent.h>.

B.4 Open on a Directory

Some **old** examples use the function open() to recognize if the file is a directory or not. If the open() is applied on a directory, its return is -1 and the global variable **errno** is set to **EISDIR**.

The AS/400 open API attempts to conform to the POSIX 1003.1 standard. In this standard, the definition of EISDIR for open() states that it can be returned when the oflag specifies O_RDWR or O_WRONLY and the object is a directory. This means that a directory cannot be opened for writing. But, it implies that a directory can be opened for reading. Therefore, the source code should be changed in some way; either the inline macro S_ISDIR is used or the function opendir() should be used.

```
int     isdir(char *path)
{
#ifdef __ILEC400__
    struct stat stbuf;

    if (stat(path, &stbuf) == -1) {
        perror ("stat");
        exit (-1);
    }
    return (S_ISDIR(stbuf.st_mode));         1
#else
    int  dirf;
    int  fd;
    /* dirf is true if the parameter path cannot be opened and
```

```
     * the global variable errno is set with EISDIR.
     */
    dirf = (fd = open(path, 1)) == -1 && errno == EISDIR; 2
    if (fd != -1 && close(fd) == -1) {
        perror ("close");
        exit (-1);
    }
    return ans;
#endif
}
```

The following source code shows how nice it is changed. Because some implementations are dependent on each machine, it is recommended to use macro directives and cutout the machine or OS dependencies. In the following source code, the macro directive #ifdef can even be omitted because the function opendir() is standard function and, therefore, the user function isdir() is not necessary.

```
#ifdef __ILEC400__
#define isdir(aDir)    (opendir(aDir) != NULL)  3
#else
int     isdir(char *path)
{
   ...
}
#endif    /* End of __ILEC400__ */
```

Notes:

1 See the previous example and note.

2 The function open(..,1) is used on a directory with the parameter 1, which means read only. It should be used as open(..,O_RDONLY) and after the POSIX definition, this function returns 0.

3 It shows the better way to port old-style C source codes.

B.5 Link on a Directory

A simple way to move an object into another directory is by copying the object to another object and deleting the original object. But to copy a directory is difficult and this takes much time. This method is not elegant at all. A better way is to link and unlink.

In this example, the functions link() and unlink() are used to move a file or directory to another file or into a directory.

```
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>

#define  isdir(aDir)      (opendir(aDir) != NULL)

int     move (const char *srcobj, const char *destobj)
{
    char  destdir [MAXLEN]
    .
    .
    strcpy (destdir, destobj);

    if (!(dirf = isdir(destdir))) {
        /* the object is not a directory but a regular file.
         * should be deleted
         */
        if (unlink(destdir) == -1 && errno != ENOENT) {
            perror ("unlink");
            exit (-1);
        }
    } else {
       /* the destdir is a directory. the source name will be
        * appended at the end of destdir.
        */
        char   *p = strrchr(srcobj, '/');
        if (p == NULL)
            p = srcdir;
        else
            p++;
        strcat (destname, "/");
        strcat (destname, p);
    }
    .
    .
    /* link the object
    */
    if (link (srcobj, destname) == -1) {    1
        perror ("link");
        exit (-1);
    }
    .
    .
    /* unlink the source object
    */
    if (unlink (srcobj) == -1) {
        perror ("unlink");
        exit (-1);
    }
}
```

Notes:

1 This function returns fail on integrated file system if it is applied on a directory. The function link(aFile, aDir) returns 0 as successful. However, link(aDir,aDestDir) returns -1 as fail on the integrated file system.

Links created by this function are not allowed to cross file systems. For example, you cannot create a link to a file in the QOpenSys directory from the root directory.

B.6 Access of Global Variable sys_errlist, nsyserr

Even though the global variables **char *sys_errlist??(??)** and **int sys_nerr** are not documented in the POSIX definition, the preceding variable is accessed in the UNIX world as usual in order to describe an error message in a magical way.

For example, the user can attempt to open a certain file to write, which is a directory or cannot be accessed to write by permissions. Naturally, this file cannot be opened and the global variable **errno** is set with a value of either **EACCES** or **EISDIR**. The function perror() shows the corresponding error message on the display. However, the user wants to modify this error message and display it more beautifully while accessing the global variable sys_nerr and sys_errlist.

The following source code shows how the preceding variables are accessed in the UNIX system. This does not work on the integrated file system.

```
   .
   .
#include <errno.h>
  ...
      /* declaration of global variables */
      extern int sys_nerr;
      extern char *sys_nerr·‘;
  ...
      /* checkinh the valid range of errno */
      if (errno > 0 && errno < sys_nerr)       1
          p = sys_errlist·errno‘; /* return the corresponding */   2
                                  /* text pointer             */
```

Notes:

1 sys_nerr is the maximum number of the list of error text.

2 sys_errlist is an array of the pointer to the error texts.

This is not portable to the AS/400 system. Because, as previously mentioned, the integrated file system is designed to confirm the POSIX definition, the user should use the integrated file system APIs with the POSIX definitions. The following program code, which is provided by the integrated file system developers, simulates the way to get the error message text pointer.

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <qmhrtvm.h>   /* retrieve message SPI and types */
#include <qusec.h>     /* SPI Error Code Parameter types */

char *sys_errlist(int errnovalue);

int main(int argc, char *argv??(??))
{
   char                *p = NULL;

   if (argc != 2) {
     printf("Usage:  syserr <errnovalue>\n");
     exit(1);
   }

   errno = atoi(argv??(1??));
   p = sys_errlist(errno);
   if (p != NULL) {
      printf(”Error text <%s>\n”, p);
      free(p);
      p = NULL;
   }

   return;
}


/********************************************************************/
/* char *sys_errlist(int errnovalue)                                */
/*                                                                  */
/* Return a pointer to a character string representing              */
/* the error that caused the particular errno value.                */
/*  NOTE: The string returned should be free′d by the caller.       */
/*        If the errno is invalid a string stating this is returned */
/********************************************************************/
char *sys_errlist(int errnovalue)
{

  typedef _Packed struct {
     Qmh_Rtvm_RTVM0100_t mdata;        /* Basic Message Data */
     char                mtext·1024‘;  /* Kind of arbitrary, enough? */
  } MyMessage_t;

  /* Information about the message that we′re going to retrieve */
  char         msgformat??(9??)  = ”RTVM0100”;
  char         msgid??(8??)      = ”CPExxxx”;
  char         msgfile??(21??)   = ”QCPFMSG   QSYS      ”;
  char        *msgsubstdta       = NULL;   /* we won′t be substituting here */
  int          msgsubstlen       = 0;
  char         msgsubst??(11??)  = ”*NO       ”;
  char         msgfmtctl??(11??) = ”*NO       ”;
  Qus_EC_t     errorcode;
  char        *p             = NULL;    /* string to return              */
  MyMessage_t  message;                 /* message we′ll retrieve        */
  char        invalstring·“  = ”Invalid ERRNO value”;

  /* ERRNO messages are in the range of CPE0000 to CPE9999               */
  if (errnovalue < 0 || errnovalue > 9999) {
    return NULL;
  }
  sprintf(msgid+3, "%0.4d", errnovalue);
  printf("Getting MSGID=<%s>\n", msgid);  /* INFORMATIONAL */

  memset(&errorcode, 0, sizeof(errorcode));    /* Clear the error area   */
  errorcode.Bytes_Provided = sizeof(errorcode);/* Init the error code    */

  memset(&message, 0, sizeof(message.mdata));  /* Clear the message area */
  message.mtext??(0??)='\0';                   /* Null Text String       */

  QMHRTVM( &message,
           sizeof(message),
           msgformat,
           msgid,
           msgfile,
           msgsubstdta,
           msgsubstlen,
           msgsubst,
           msgfmtctl,
           &errorcode);
  if (errorcode.Bytes_Available) {
    printf("Error on retrieve message: <%.7s>\n",
           errorcode.Exception_Id);  /* INFORMATIONAL */
    p = (char *)malloc(sizeof(invalstring));
    memcpy(p, invalstring, sizeof(invalstring));
  } else {
    p = (char *)malloc(message.mdata.Length_Message_Returned + 1);
    memcpy(p, message.mtext, message.mdata.Length_Message_Returned);
    p·message.mdata.Length_Message_Returned‘ = '\0';
  }

  return p;
}
```

Appendix C. Development Cycle of ILE C/400 Applications

Development of a program is a cycle work of editing, compiling, binding, and debugging. ILE (Integrated Language Environment) on the AS/400 system provides applications development (and execution) environment from the program creation phase through the debugging phase.

Figure 100 on page 260 illustrates the typical stages in developing C applications in the ILE environment.

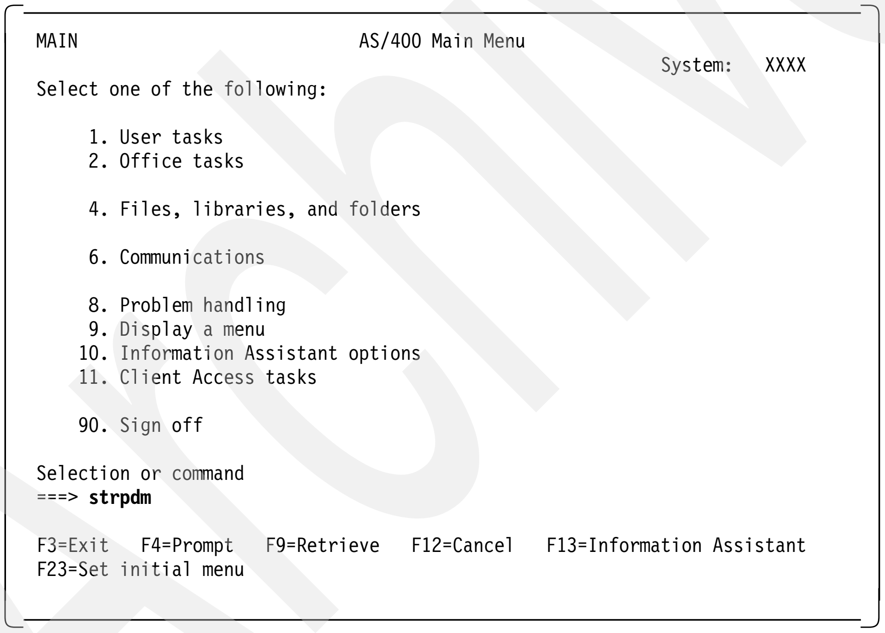
```
Application Development Stages                  CL Commands

1. Code your          @=============#           Start Source
   source program     |             |           Entry Utility
                      |    Source   |           (STRSEU)
                      |     Code    |<========#
                      |             |         |
                      $======+======%         |
                             |                |
==============================+================+================
                             |                |
                             V                |
2. Compile your       @=============#         |  Create a
   source program     |             |         |  Module
                      |    Module   |         |  (CRTCMOD)
                      |    Object   |         |
                      |             |         |
                      $======+======%         |
                             |                |
==============================+================+================
                             |                |
                             V                |
3. Create the         @=============#         |  Create a
   program object     |             |         |  Program
   by binding one     |    Program  |         |  (CRTPGM)
   or more modules    |    Object   |         |  (CRTSRVPGM)
                      |             |         |
                      $======+======%         |
                             |                |
==============================+================+================
                             |                |
4. Enable OS/400             |                |  Start Debug
   debug mode                |                |  (STRDBG)
                             |                |
==============================+================+================
                             |                |
                             V                |
5. Run your           @=============#         |  Call a
   program            |             |         |  Program
                      |    Program  |         |  (CALL)
                      |    Runs     |         |
                      |             |         |
                      $======+======%         |
                             |                |
==============================+================+================
                             |                |
                             V                |
6. Use source         @=============#         |
   debugger           |             |         |
                      |    Debug    |         |
                      |    Mode     "=========%
                      |             |
                      $=============%
```

Figure 100. Stages in C Application Development

C.1 Source Editing

If your UNIX source codes are imported into members of a source physical file, you can modify the existing source code or edit a new source code with the editor in the Programming Development Manager tool. Let′s assume that your source code ′MYPGM′ is in ′QCSRC′ file of ′MYLIB′ library. This is the same as with the ′/MYLIB/QCSRC′ directory in the UNIX system.

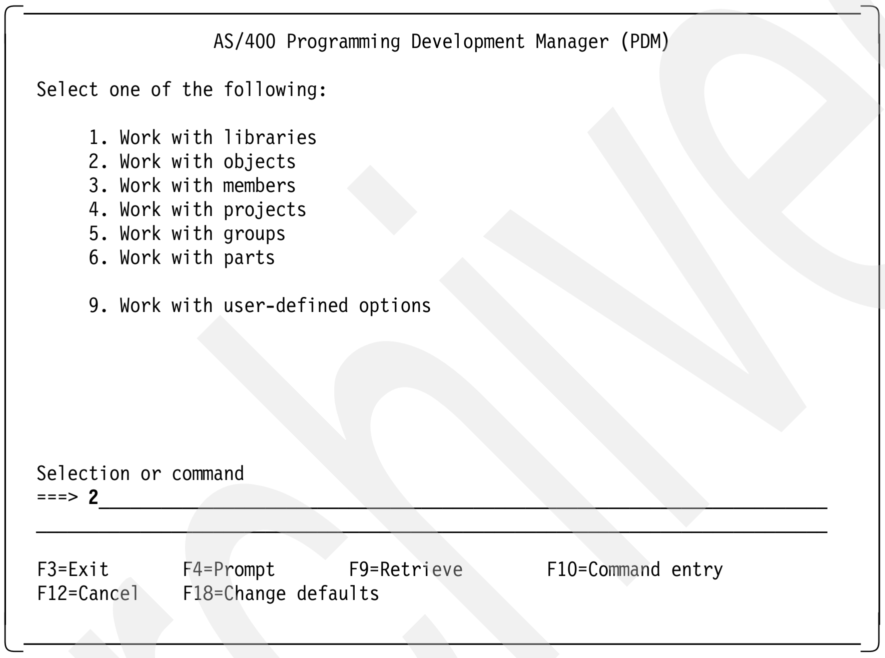
To start Programming Development Manager, you enter **STRPDM** on the command line and press Enter. Then the AS/400 Programming Development Manager menu is displayed.



```
MAIN                           AS/400 Main Menu
                                                            System:   XXXX
Select one of the following:

     1. User tasks
     2. Office tasks

     4. Files, libraries, and folders

     6. Communications

     8. Problem handling
     9. Display a menu
    10. Information Assistant options
    11. Client Access tasks

    90. Sign off

Selection or command
===> strpdm

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu
```

Figure 101. AS/400 Main Menu with STRPDM Command Specified

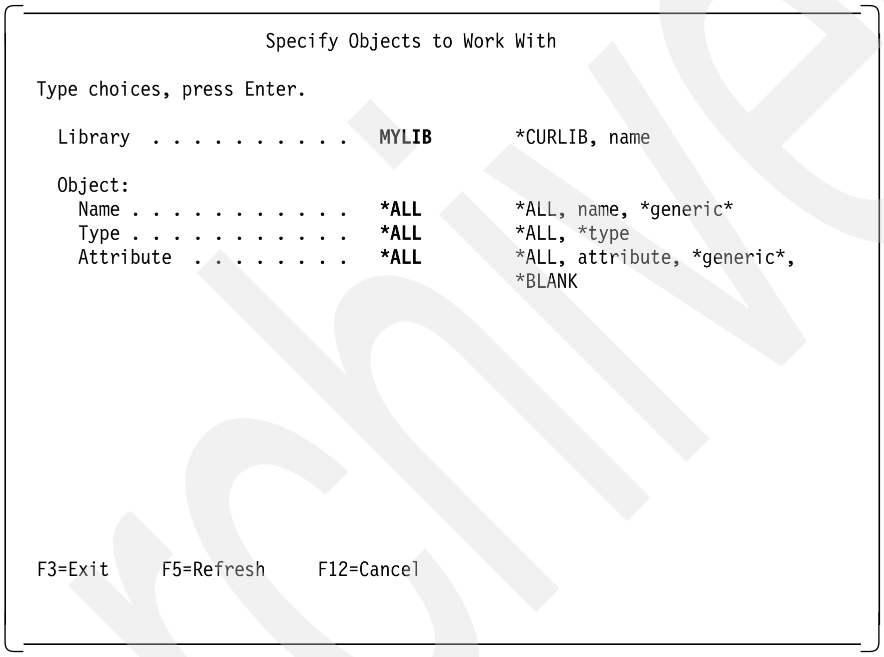
Select **option 2** and press Enter. Then the Specify Objects to Work With is displayed. You can use this menu to select the type of list you want to work with.



```
                 AS/400 Programming Development Manager (PDM)

 Select one of the following:

      1. Work with libraries
      2. Work with objects
      3. Work with members
      4. Work with projects
      5. Work with groups
      6. Work with parts

      9. Work with user-defined options





 Selection or command
 ===> 2
 ____________________________________________________________________________

 F3=Exit       F4=Prompt       F9=Retrieve        F10=Command entry
 F12=Cancel    F18=Change defaults
```

Figure 102. AS/400 Programming Development Manager (PDM) Menu with Option 2 Specified

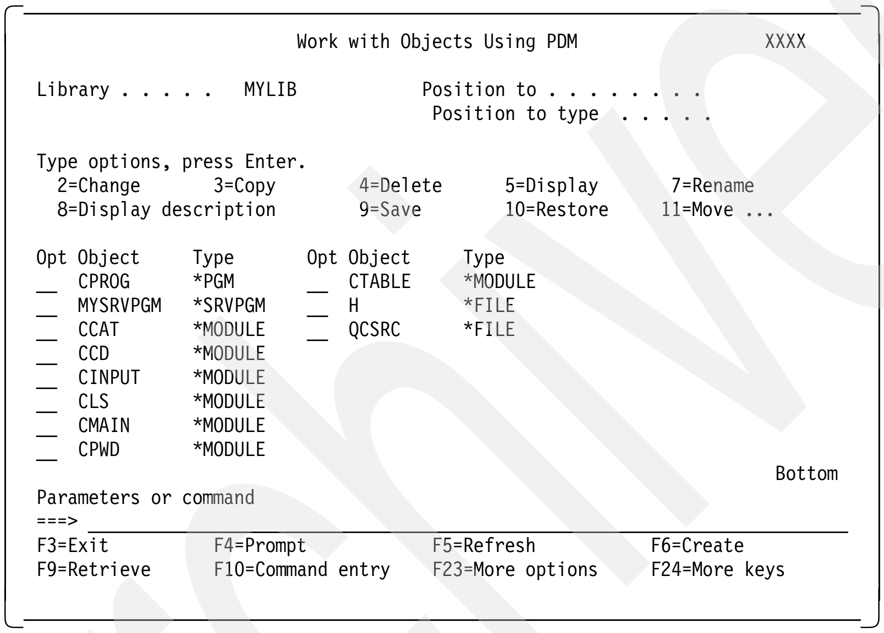
You can use the Specify Objects to Work With window to select the objects you want to work with. You can use the Library, Name, Type, and the Attribute prompts to specify subset criteria. When you enter ′**MYLIB**′ and press Enter, the Work with Objects Using PDM menu for ′MYLIB′ is displayed.



```
                       Specify Objects to Work With

 Type choices, press Enter.

   Library  . . . . . . . . . .   MYLIB        *CURLIB, name

   Object:
     Name . . . . . . . . . . . .   *ALL         *ALL, name, *generic*
     Type . . . . . . . . . . . .   *ALL         *ALL, *type
     Attribute  . . . . . . . . .   *ALL         *ALL, attribute, *generic*,
                                                 *BLANK










 F3=Exit     F5=Refresh     F12=Cancel
```

Figure 103. Specify Objects to Work With Window

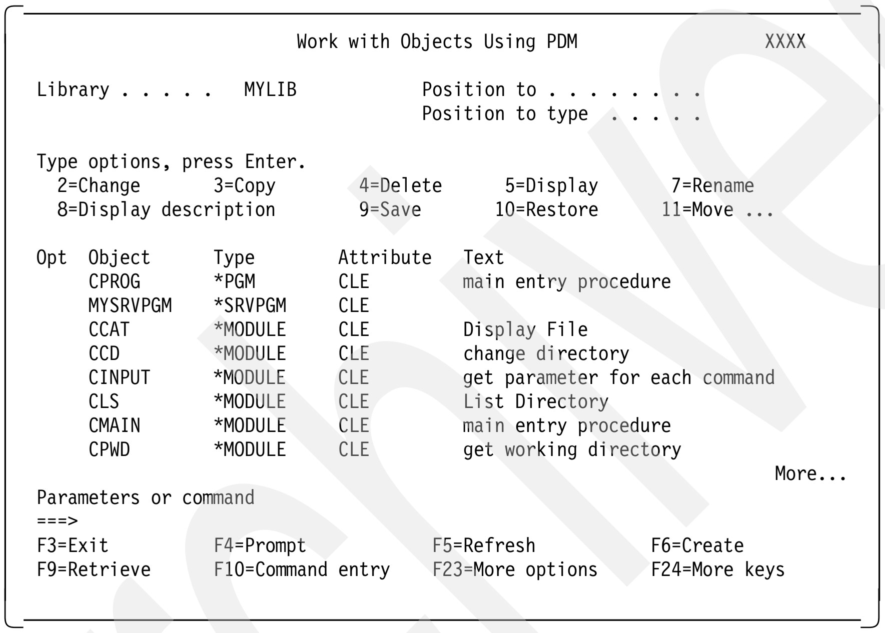
You can use this menu to work with objects in a library by selecting options or pressing function keys. Object name ′QCSRC′ is an AS/400 naming convention and usually this file has C source members in it.



```
                         Work with Objects Using PDM                  XXXX

Library . . . . .   MYLIB           Position to . . . . . . . .
                                     Position to type  . . . . .

Type options, press Enter.
  2=Change       3=Copy        4=Delete      5=Display       7=Rename
  8=Display description        9=Save        10=Restore     11=Move ...

Opt Object     Type       Opt Object     Type
__  CPROG      *PGM       __  CTABLE     *MODULE
__  MYSRVPGM   *SRVPGM    __  H          *FILE
__  CCAT       *MODULE    __  QCSRC      *FILE
__  CCD        *MODULE
__  CINPUT     *MODULE
__  CLS        *MODULE
__  CMAIN      *MODULE
__  CPWD       *MODULE
                                                                          Bottom
Parameters or command
===> ___________________________________________________________________________
F3=Exit          F4=Prompt             F5=Refresh            F6=Create
F9=Retrieve      F10=Command entry     F23=More options      F24=More keys
```

Figure 104. Work with Objects Using PDM

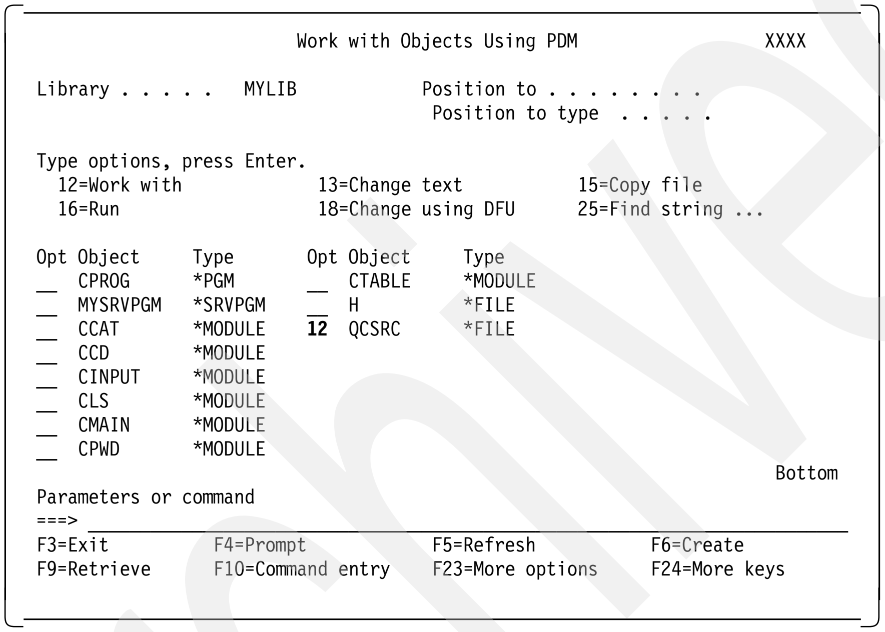
If you want to see the Attribute field and Text field, then press the <F11> key. When you press <F11> again, the list is changed to the original form. This is almost the same with 'DIR/W' and 'DIR/P' in DOS or the OS/2 system.



```
                         Work with Objects Using PDM                XXXX

 Library . . . . .   MYLIB            Position to . . . . . . . .
                                      Position to type  . . . . .

 Type options, press Enter.
   2=Change      3=Copy        4=Delete      5=Display      7=Rename
   8=Display description       9=Save       10=Restore      11=Move ...

 Opt  Object      Type        Attribute   Text
      CPROG       *PGM        CLE         main entry procedure
      MYSRVPGM    *SRVPGM     CLE
      CCAT        *MODULE     CLE         Display File
      CCD         *MODULE     CLE         change directory
      CINPUT      *MODULE     CLE         get parameter for each command
      CLS         *MODULE     CLE         List Directory
      CMAIN       *MODULE     CLE         main entry procedure
      CPWD        *MODULE     CLE         get working directory
                                                                   More...
 Parameters or command
 ===>
 F3=Exit          F4=Prompt            F5=Refresh           F6=Create
 F9=Retrieve      F10=Command entry    F23=More options     F24=More keys
```

Figure 105. Work with Objects Using PDM

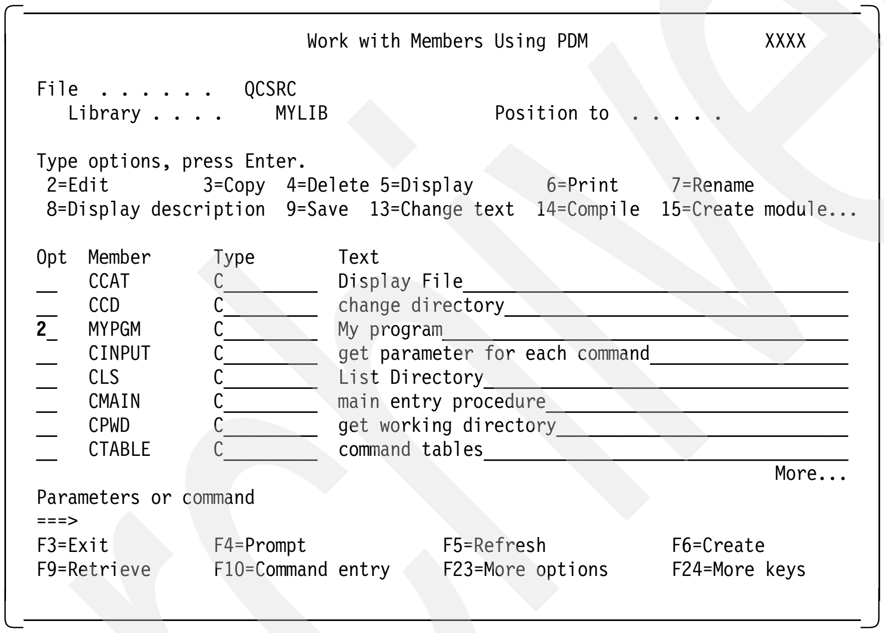
To work with members of 'QCSRC' file, you enter **option 12** on the 'QCSRC' object. You can see option 12 by pressing the F23 key.



```
                         Work with Objects Using PDM                  XXXX

 Library . . . . .   MYLIB           Position to . . . . . . . .
                                      Position to type  . . . . .

 Type options, press Enter.
   12=Work with            13=Change text           15=Copy file
   16=Run                  18=Change using DFU      25=Find string ...

 Opt Object     Type       Opt Object     Type
 __  CPROG      *PGM       __  CTABLE     *MODULE
 __  MYSRVPGM   *SRVPGM    __  H          *FILE
 __  CCAT       *MODULE    12  QCSRC      *FILE
 __  CCD        *MODULE
 __  CINPUT     *MODULE
 __  CLS        *MODULE
 __  CMAIN      *MODULE
 __  CPWD       *MODULE
                                                                       Bottom
 Parameters or command
 ===> ____________________________________________________________________
 F3=Exit          F4=Prompt            F5=Refresh           F6=Create
 F9=Retrieve      F10=Command entry    F23=More options     F24=More keys
```

Figure 106. Work with Objects Using PDM Menu with Option 12 Specified

When you enter 12 in the Opt column by ′QCSRC′, members of the source physical file QCSRC are listed. You can use this menu to work with members in a physical file by selecting options or pressing function keys. When you enter **option 2** on ′MYPGM′ and press Enter, the Source Entry Utility(SEU) window is displayed.



```
                          Work with Members Using PDM                XXXX

 File  . . . . . .   QCSRC
    Library . . . .     MYLIB              Position to  . . . . .

 Type options, press Enter.
  2=Edit         3=Copy  4=Delete 5=Display       6=Print     7=Rename
  8=Display description  9=Save  13=Change text  14=Compile  15=Create module...

 Opt  Member      Type        Text
 __   CCAT        C_________  Display File____________________________________
 __   CCD         C_________  change directory________________________________
 2_   MYPGM       C_________  My program______________________________________
 __   CINPUT      C_________  get parameter for each command__________________
 __   CLS         C_________  List Directory__________________________________
 __   CMAIN       C_________  main entry procedure____________________________
 __   CPWD        C_________  get working directory___________________________
 __   CTABLE      C_________  command tables__________________________________
                                                                         More...
 Parameters or command
 ===>
 F3=Exit          F4=Prompt             F5=Refresh            F6=Create
 F9=Retrieve      F10=Command entry     F23=More options      F24=More keys
```

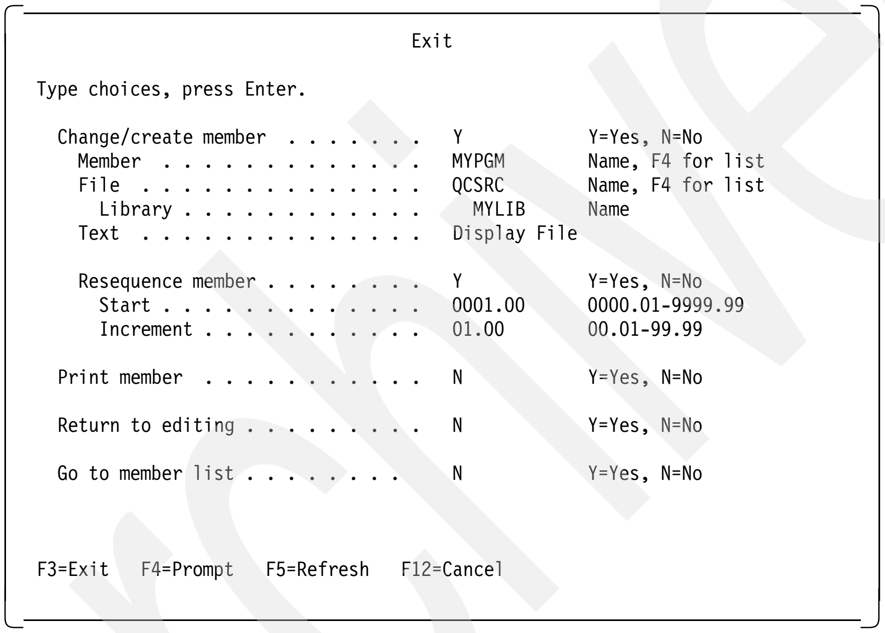
Figure 107. Work with Members using PDM

You can edit your source code in the SEU window. After editing, you can exit SEU by pressing the F3 key. When you press F3, the Exit Confirmation window is displayed.

```
 Columns . . . :    1  80                                      Edit
 SEU==>
 FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 .
        *************** Beginning of data *************************************
0001.00 #include <stdlib.h>
0002.00 #include <stdio.h>
0003.00 #include <fcntl.h>
0004.00 #include <unistd.h>
0005.00 #include <sys/types.h>
0006.00 #include <errno.h>
0007.00
0008.00 #include "hcomm.h"
0009.00
0010.00 int      c_cat(int argc, char *argv??(??))
0011.00 {
0012.00     int      fd;
0013.00     int      BytesRead;
0014.00     char     buffer??(BUFFSIZE??);
0015.00     char     *filename = argv??(0??);
0016.00
0017.00     if ((fd = open(filename, O_RDONLY)) == -1)
0018.00         return (errno);
0019.00

 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F10=Cursor
 F16=Repeat find       F17=Repeat change          F24=More keys
```

Figure 108. Source Entry Utility (SEU)

You can save, print a member, or return to the SEU window on the Exit Confirmation window. When you press Enter with the default option, the edited source code is saved and ′Member MYPGM in file MYLIB/QCSRC is changed with xxx records. A message is displayed on the Work with Members Using PDM menu.



```
                                    Exit

 Type choices, press Enter.

   Change/create member  . . . . . . .   Y            Y=Yes, N=No
     Member  . . . . . . . . . . . . .   MYPGM        Name, F4 for list
     File  . . . . . . . . . . . . . .   QCSRC        Name, F4 for list
       Library . . . . . . . . . . . .     MYLIB      Name
     Text  . . . . . . . . . . . . . .   Display File

     Resequence member . . . . . . . .   Y            Y=Yes, N=No
       Start . . . . . . . . . . . . .   0001.00      0000.01-9999.99
       Increment . . . . . . . . . . .   01.00        00.01-99.99

   Print member  . . . . . . . . . . .   N            Y=Yes, N=No

   Return to editing . . . . . . . . .   N            Y=Yes, N=No

   Go to member list . . . . . . . .     N            Y=Yes, N=No



 F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel
```

Figure 109. Exit Confirmation

C.2 Module Creation

Because most C applications consist of many source codes on the UNIX system, the UNIX system provides a make tool to build a program. With ILE C/400, each module can be compiled separately and all modules can be bound into a program. The ILE C/400 provides a make utility ″TMKMAKE″ besides compilers and binders. For a big project, some developer creates his own object libraries. But the ILE C/400 does not provide library facilities, however, runtime libraries called ″SERVICE PROGRAM″ can be created and used easily. The following list can help the developer who ports a UNIX application to the the AS/400 system.

Creating a module is very similar to compiling a source code without creating a program in the UNIX system.

In UNIX, you can compile a source code with the command

```
cc -c prog.c.
```

On the AS/400 system, you can create a module from a C source code with the command **CRTCMOD**.

The CRTCMOD command starts an ILE C compiler which, upon successful compilation of the source code, creates a permanent module object that can be optionally combined with other modules into a bound program. As in the preceding syntax diagram, the command has a set of parameters. The following parameters are relevant to the runtime environment of your program:

DBGVIEW	Specifies the option that controls which views of the input source or generated listing are available for debugging the compiled module.
OPTIMIZE	Specifies the optimization level of the module.

If you want to change the optimization level of the code or change the debugging ability at runtime, you must set the appropriate parameters at compile time.

Table 19 shows related CL commands to Create C Module (CRTCMOD).

Table 19. Related CL Command to Create C Module (CRTCMOD)

CL Command	Description
CRTRPGMOD	Create RPG Module
CRTCBLMOD	Create COBOL Module
DSPMOD	Display Module
CHGMOD	Change Module
DLTMOD	Delete Module
WRKMOD	Work with Module

C.3 Creating/Binding an ILE Program

In UNIX, you can create or bind a program with the command:

```
ld -o progname ...
```

On the AS/400 system, you can create or bind a program from modules with the command **CRTPGM**.

C.3.1 Creating an ILE Program from One or More Modules

For example, there are two modules to be bound into a program, MAIN and PROG.

```
CRTPGM PGM(MYPROG) MODULE(MAIN PROG)
```

When you enter the preceding command, the OS/400 program creates the program object in the following way.

1. Copies the listed modules into what becomes the program object.
2. Finds the module with the program entry procedure.
3. Locates the first import in the module with the program entry procedure.
4. Checks the modules in the order listed in the command until the first import is matched with a module export.
5. Returns to the first module to find the next import.
6. Resolves all imports in the first module, then goes to the next module in the list.
7. Resolves all imports in the second module.
8. Goes to each subsequent module in the list until all imports are resolved.
9. If one or more imports remain without an export, the binding ends without creating the program object.
10. When all imports have been resolved, the binding ends and the program object exists with the specified library and name.

Table 20 on page 272 shows related CL commands to Create Program (CRTPGM).

Table 20. Related CL Command to Create Program (CRTPGM)

CL Command	Description
CRTBNDRPG	Create Bound RPG Program
CRTBNDCBL	Create Bound CBL Program
DSPPGM	Display Program
CHGPGM	Change Program
DLTPGM	Delete Program

C.3.2 Service Program

A service program is a special kind of ILE program. Service programs are like subroutine libraries. They are also like dynamic link libraries (DLLs) in the OS/2 system or like shared libraries in the UNIX system. They are used for common functions that are frequently called within an application.

Program can be created without service programs. But if an application is very large, it is very difficult to maintain a program. In deciding whether to use service programs, you should weigh their advantages and disadvantages. Table 21 on page 273 shows the advantages and disadvantages of service programs.

An ILE service program is created by binding one or more ILE objects together to make one functional package such as a shared library in the UNIX system. The objects bound together must include at least one module and can include other service programs also. It has an object type of *SRVPGM to distinguish it from other bound programs.

A service program provides you with a means of packaging externally supported callable routines (functions) into a separate object.

To create a service program, use the Create Service Program (**CRTSRVPGM**) command. This command binds one or more modules into a runnable object by using a process similar to the process used by the **CRTPGM** command. The modules that are packaged together in the service program do not need to have any functional relationship to each other. The only requirement for the object to be created is that all imports must be resolved and, for the program to be useful, some exports are defined for use by other ILE programs.

<<syntax of CRTSRVPGM>>

Table 21. Advantages and Disadvantages of using the Service Program

Advantages	Disadvantages
Simpler maintenance of applications	Resolving system pointers for additional service programs.
Less time to build applications	Activation time for the service programs, including allocation of the activation, runtime binding, and static initialization.
Hidden information	Application build time
Less memory and disk space	Parts management

Table 22 shows related CL commands to Create Program (CRTPGM).

Table 22. Related CL Command to Create Service Program (CRTSRVPGM)

CL Command	Description
DSPSRVPGM	Display Service Program
CHGSRVPGM	Change Service Program
DLTSRVPGM	Delete Service Program

C.4 Debugging an ILE Program

For the program developer, it is very hard to find an error in a program. If you saves time on debugging, the whole develop time is reduced and then the development cost is less. The ILE source debugger is used to help find programming errors in ILE programs and service programs.

Before you can use the ILE source debugger. you must use the debug option (**DBGVIEW**) when you create a module object (**CRTCMOD**) or a program object. Then you can start your debug session. Once you set breakpoints or other ILE source debugger options, you can call the program.

You use the Start Debug (**STRDBG**) command to start the ILE source debugger. Once the debugger is started. it remains active until you enter the End Debug (**ENDDBG**) command.

Table 23 shows related CL commands to Start Debug (STRDBG).

Table 23. Related CL Command to Start Debug (STRDBG)

CL Command	**Description**
ENDDBG	End Debug
DSPMODSRC	Display Module Source

C.5 Tour of Development Cycle with Real Examples

This section shows user the entire development cycle with sample source codes: how to create an application in ILE environment and how to debug errors.

This program is simple enough, we hope, to be self-explanatory. This program displays a menu and the current working directory in the integrated file system, reads the user input and runs the corresponding functions (display list, change current working directory, and display file). The program consists of an ILE program and a service program.

C.5.1 List of Source Codes

The following source codes are in ′QCSRC′ and ′H′ files in ′MYLIB′ library.

.

- ccomm.h -- common header file
- cproto.h -- prototype definitions
- cmain.c -- main entry function
- ctable.c -- function table definitions
- cinput.c -- input functions
- ccat.c -- display of file
- cls.c -- display of directory
- cpwd.c -- current working directory
- ccd. -- change working directory

C.5.2 Source Codes

```
#ifndef __CCOMM__
#define __CCOMM__

#define  COK       0
#define  CEXIT     9

#define  MAXCMD    3
#define  MAXARG    3
#define  BUFFSIZE  1024
#define  MAXNAMLEN 1024

extern   int   (*cmdlist[])(int argc, char **argv);
extern   int   (*getlist[])(char **argv);
extern   char  *cmdname[];
extern   char  *curwrkdir;

#endif   /* __CCOMM__ */
```

Figure 110. ccomm.h

```
#ifndef __CPROTO__
#define __CPROTO__

int    c_ls  (int arg, char *argv[]);
int    c_pwd (int arg, char *argv[]);
int    c_cd  (int arg, char *argv[]);
int    c_cat (int arg, char *argv[]);

int    c_ls_get  (char *argv[]);
int    c_pwd_get (char *argv[]);
int    c_cd_get  (char *argv[]);
int    c_cat_get (char *argv[]);
#endif /*__CPROTO__*/
```

Figure 111. cproto.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include ″ccomm.h″
#include ″cproto.h″

static  int   narg;
static  char  *argp[MAXARG];
static  char  argbuf[BUFFSIZE*MAXARG];

static void print_menu(void);
static int  get_func(void);

static void print_menu(void)
{
    printf (″\n″);
    printf (″Current Working Directory: %s\n\n″, curwrkdir);
    printf (″1. List Directory\n″);
    printf (″2. Change Working Directory\n″);
    printf (″3. Display File\n″);
    printf (″\n\n″);
    printf (″9. Exit\n″);
}
```

Figure 112. cmain.c - 1 of 2

```
static int get_func(void)
{
    int   ret;
    char  ans[128];

    while (1) {
        print_menu();
        gets(ans);
        ret = atoi (ans);
        if (ret <= 0 || ret > MAXCMD) {
            if (ret == CEXIT)
                return CEXIT;
            fprintf (stderr,"Invalid Choice!!!\n");
            continue;
        }
        narg = (*getlist[ret-1])(argp);
        break;
    };
    return ret;
}


int    main ()
{
    int    i;
    int    ret;

    for (i = 0; i < MAXARG ; i++)
        argp[i] = &argbuf[i*BUFFSIZE];

    while (1) {
        if ((ret = c_pwd(narg, argp)) != COK) {
            perror("Current Working Directory");
            break;
        }
        if ((i = get_func()) != CEXIT) {
            if((ret = (*cmdlist[i-1])(narg, argp)) != COK)
                perror(cmdname[i-1]);
        } else
            break;
    }
}
```

Figure 113. cmain.c - 2 of 2

```
#include <stdio.h>
#include <stdlib.h>

#include ″cproto.h″

int     c_ls_get (char *argv[])
{
     printf (″ Enter the directory name\n″);
     gets(argv[0]);
     return 1;
}

int     c_cd_get (char *argv[])
{
     printf (″ Enter the directory name\n″);
     gets(argv[0]);
     return 1;
}

int     c_cat_get (char *argv[])
{
     printf (″ Enter the File name\n″);
     gets(argv[0]);
     return 1;
}
```

Figure 114. cinput.c

```
#include "cproto.h"

int    (*cmdlist[])(int argc, char *argv[]) = {
       c_ls,
       c_cd,
       c_cat
};

int    (*getlist[])(char *argv[]) = {
       c_ls_get,
       c_cd_get,
       c_cat_get
};

char   *cmdname[] = {
       "List Directory",
       "Change Working Directory",
       "Display File"
};
```

Figure 115. ctable.c

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>

#include "ccomm.h"

int      c_cat(int argc, char *argv[])
{
    int      fd;
    int      BytesRead;
    char     buffer[BUFFSIZE];
    char     *filename = argv[0];

    if ((fd = open(filename, O_RDONLY)) == -1)
        return (errno);

    while ((BytesRead = read (fd, buffer, BUFFSIZE)) > 0) {
        fwrite (buffer, 1, BytesRead, stdout);
    }
    fwrite("\n", 1, 2, stdout);

    close(fd);
    return COK;
}
```

Figure 116. ccat.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

#include <dirent.h>
#include "ccomm.h"

/* dirwalk: ...*/
static  int dirwalk (char *dir)
{
    char name[MAXNAMLEN];
    struct dirent    *dp;
    DIR      *dfd;

    if ((dfd = opendir(dir)) == NULL)
        return errno;

    while (( dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->d_name, ".") == 0 |
        strcmp(dp->d_name, "..") == 0)
            continue;

        if (strlen(dir) + strlen(dp->d_name) +2 > sizeof(name))
            fprintf(stderr, "dirwalk: name %s/%s to long\n",
                            dir, dp->d_name);
        else {
            struct stat stbuf;
            if (stat(dp->d_name, &stbuf) == -1)
                return errno;

            if (S_ISDIR(stbuf.st_mode))
                printf("DIR:  %s/%s\n", dir, dp->d_name);
```
Figure 117. cls.c - 1 of 2

```
            else
                printf("FILE: %s          %ld\n", dp->d_name, stbuf
.st_size);
        }
    }
    closedir(dfd);
    return COK;
}

/* fsize: ...*/
static  int  fsize(char *name)
{
    int    ret;
    struct stat stbuf;
    if (stat(name, &stbuf) == -1)
        return errno;

    if (S_ISDIR(stbuf.st_mode)) {
        if ((ret = dirwalk(name)))
            return ret;
    } else
        printf("FILE: %s          %ld\n", name, stbuf.st_size);
    printf("\n");
    return COK;
}

int     c_ls (int argc, char *argv[])
{
    if (strlen(argv[0]) == 0)
        return fsize(".");
    else
        return fsize (argv[0]);
}
```

Figure 118. cls.c - 2 of 2

```
#include <unistd.h>
#include <errno.h>

#include "ccomm.h"

char    *curwrkdir;

int      c_pwd(int argc, char **argv)
{
    static char  workdir[MAXNAMLEN];

    if (getcwd(workdir, MAXNAMLEN) == NULL)
        return errno;
    else {
        curwrkdir = workdir;
        return COK;
    }
}
```

Figure 119. cpwd.c

```
#include <errno.h>
#include <unistd.h>

#include "ccomm.h"

int      c_cd (int argc, char *argv[])
{
    char    *dirname = argv[0];

    if (chdir (dirname) == -1)
        return errno;
    else
        return COK;
}
```

Figure 120. ccd.c

C.5.3 Creating Modules

Before you create the modules, the library names in which there is source code should be in the library list. The command

```
ADDLIBLE MYLIB
```

adds the library ′MYLIB′ in the library list. The command

```
CHGCURLIB MYLIB
```

changes the library ′MYLIB′ to the current library on your session.

Let′s create modules by using **CRTCMOD** command. The following seven commands create seven modules in ′MYLIB′ library.

```
CRTCMOD MODULE(CMAIN)  OUTPUT(*PRINT) DBGVIEW(*ALL)
CRTCMOD MODULE(CTABLE) OUTPUT(*PRINT) DBGVIEW(*ALL)
CRTCMOD MODULE(CINPUT) OUTPUT(*PRINT) DBGVIEW(*ALL)
CRTCMOD MODULE(CPWD)   OUTPUT(*PRINT) DBGVIEW(*ALL)
CRTCMOD MODULE(CLS)    OUTPUT(*PRINT) DBGVIEW(*ALL)
CRTCMOD MODULE(CCD)    OUTPUT(*PRINT) DBGVIEW(*ALL)
CRTCMOD MODULE(CCAT)   OUTPUT(*PRINT) DBGVIEW(*ALL)
```

If you do not want the debug option or the print option and if you want compile the source code with optimization, you can enter the following:

```
CRTCMOD MODULE(CMAIN)  OPTIMIZE(*FULL)
```

C.5.4 Creating Service Programs

Because the functions c_pwd, c_ls, c_cd, and c_cat are independent of each other, they can be bound together in a service program.

Let′s create a service program by using the **CRTSRVPGM** command. The command:

```
CRTSRVPGM SRVPGM(MYSRVPGM) MODULE(CCD CPWD CCAT CLS) EXPORT(*ALL)
```

creates service program ′MYSRVPGM′ in ′MYLIB′ library.

C.5.5 Creating Programs

Let′s create an ILE program by using the **CRTPGM** command. The command:

```
CRTPGM PGM(CPROG) MODULE(CMAIN CINPUT CTABLE) BNDSRVPGM(MYSRVPGM)
```

creates an ILE program **CPROG** in ′MYLIB′ library.

If the current library is not set, the program is the first library in the library list that can be displayed or edited with the command:

```
EDTLIBL
```

C.5.6 Executing Programs

Let′s run the program ′CPROG′. The command:

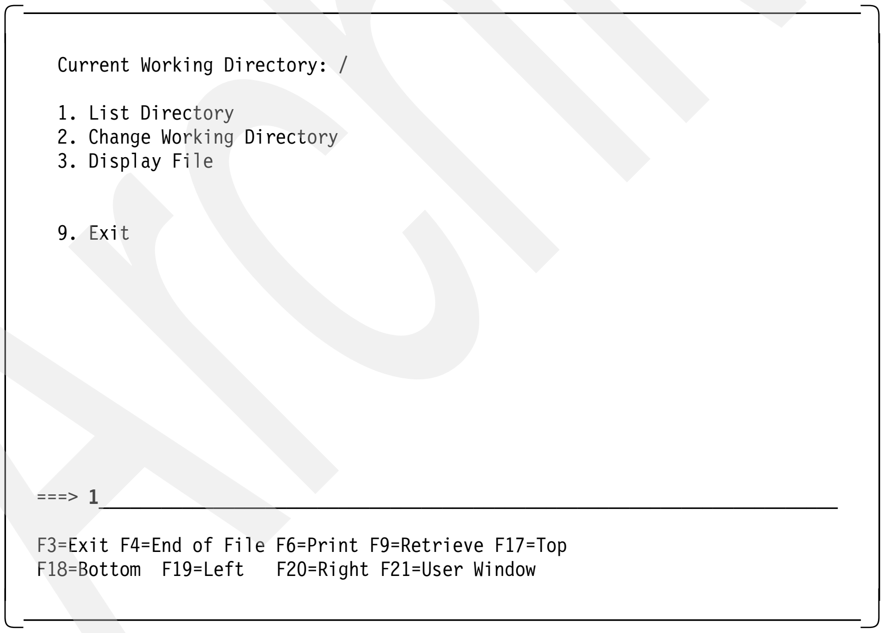
```
CALL MYLIB/CPROG
```

runs the ′MYLIB/CPROG′ program. Or, if ′MYLIB′ library is set to the current library, the command:

```
CALL CPROG
```

runs the ′MYLIB/CPROG′ program also.

When you run the ′CPROG′ program, Figure 121 is displayed.



```
  Current Working Directory: /

  1. List Directory
  2. Change Working Directory
  3. Display File


  9. Exit









===> 1

F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom  F19=Left   F20=Right F21=User Window
```

Figure 121. Run ILE Program CPROG

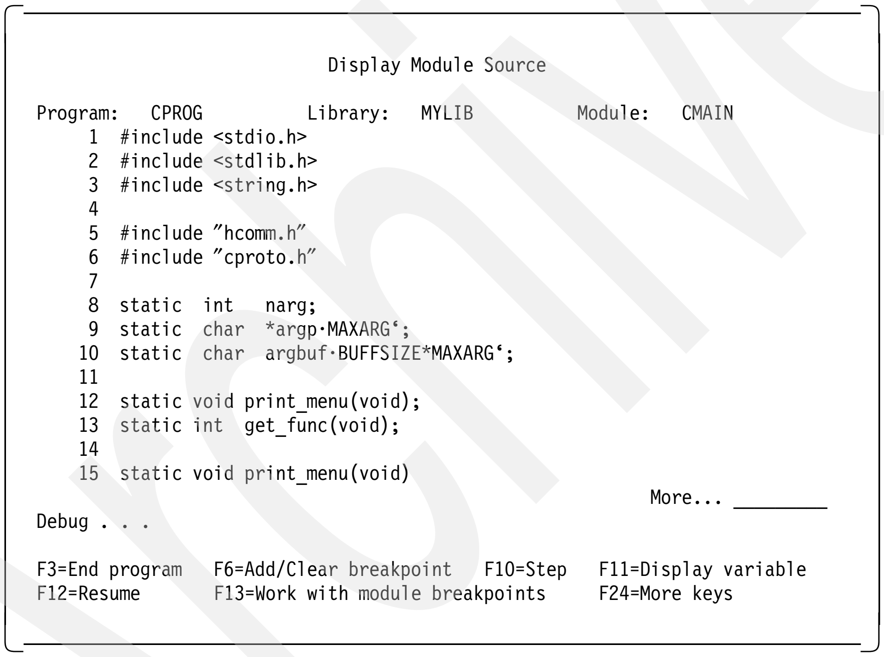
When you select **option 1** to list the directory list, it may not work because of some errors in the ′CPROG′ program.

C.5.7 Debugging Programs

Since ′CPROG′ program is created with debug option (DBGVIEW), we can debug it by using the **STRDBG** command. Let′s start debugging the ′CPROG′ program. When you enter:

```
STRDBG MYLIB/CPROG
```

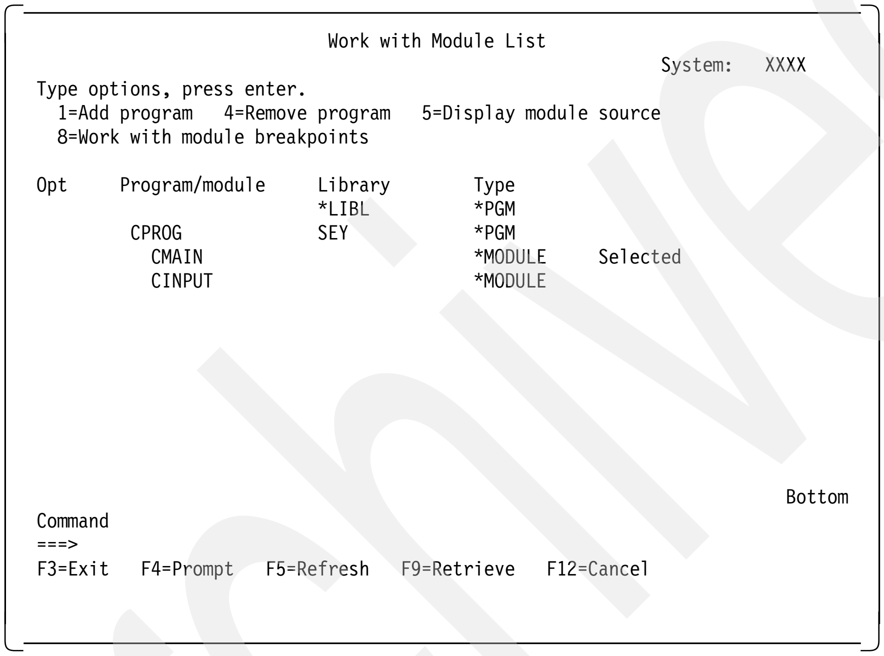
command and press Enter, the Display Module Source window for ′CMAIN′ module is displayed.



```
                            Display Module Source

 Program:   CPROG          Library:   MYLIB          Module:   CMAIN
      1  #include <stdio.h>
      2  #include <stdlib.h>
      3  #include <string.h>
      4
      5  #include ″hcomm.h″
      6  #include ″cproto.h″
      7
      8  static  int   narg;
      9  static  char  *argp·MAXARG‘;
     10  static  char  argbuf·BUFFSIZE*MAXARG‘;
     11
     12  static void print_menu(void);
     13  static int  get_func(void);
     14
     15  static void print_menu(void)
                                                             More... ________
 Debug . . .

 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F13=Work with module breakpoints     F24=More keys
```

Figure 122. Display Module Source for CMAIN Module

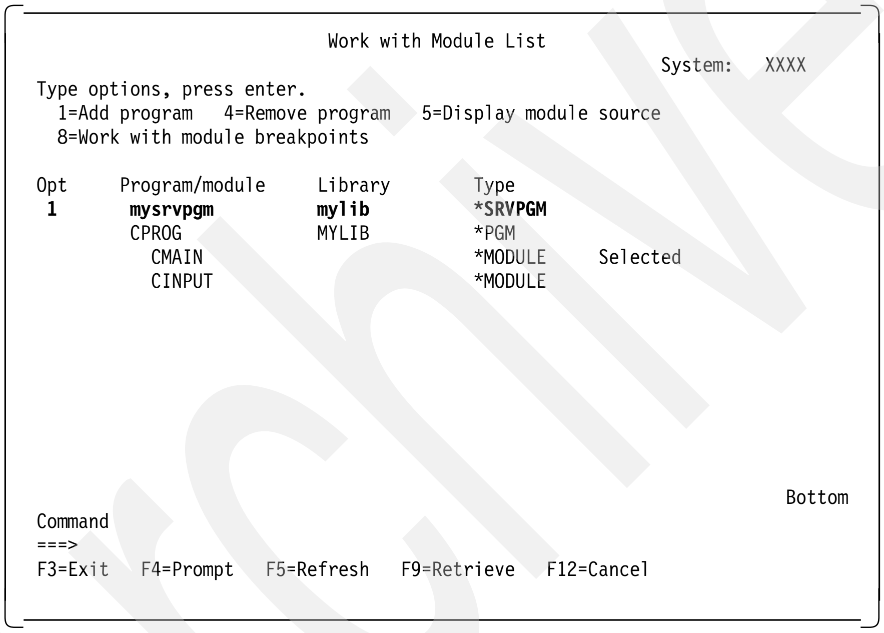
When you press the <F14> key, the Work with Module List window is displayed. You can see that the ′CMAIN′ module is selected now.



```
                             Work with Module List
                                                           System:   XXXX
 Type options, press enter.
   1=Add program   4=Remove program   5=Display module source
   8=Work with module breakpoints

 Opt     Program/module     Library        Type
                            *LIBL          *PGM
          CPROG             SEY            *PGM
            CMAIN                          *MODULE     Selected
            CINPUT                         *MODULE









                                                                      Bottom
 Command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 123. Work with Module List for CPROG Program

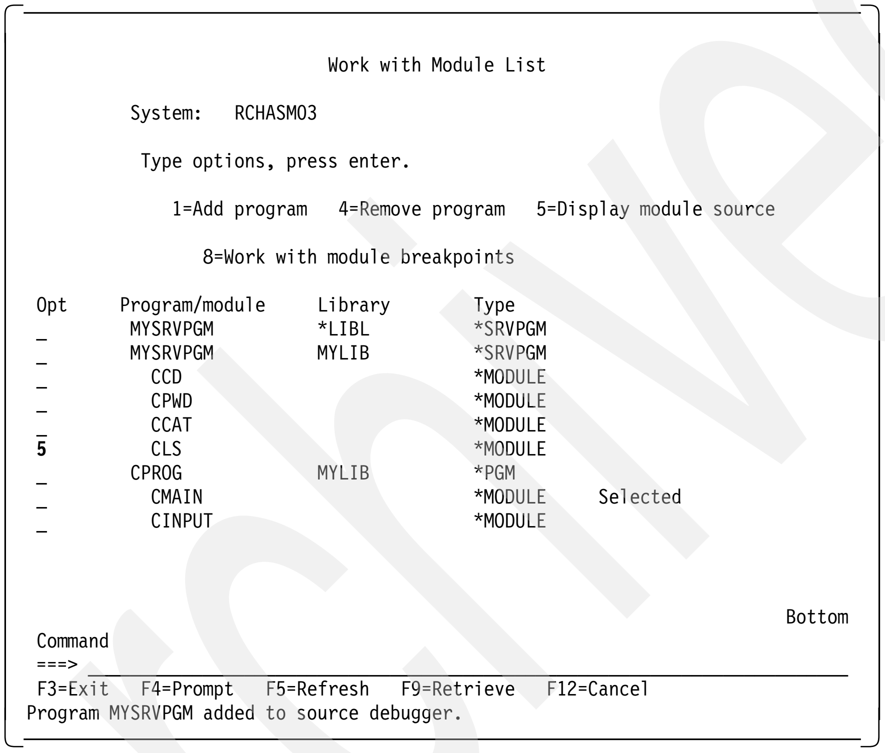
We know option 1 does not work and the source code ′CLS′ is working for option 1, so we want to see the ′CLS′ module. However, ′CLS′ is not displayed because this module is bound into the service program ′MYSRVPGM′. To debug the service program ′MYSRVPGM′, it must be added into the Program/Module list. Figure 124 shows how to add ′MYSRVPGM′ into the list.



```
                            Work with Module List
                                                            System:   XXXX
 Type options, press enter.
   1=Add program   4=Remove program   5=Display module source
   8=Work with module breakpoints

 Opt     Program/module     Library       Type
  1        mysrvpgm         mylib         *SRVPGM
           CPROG            MYLIB         *PGM
             CMAIN                        *MODULE     Selected
             CINPUT                       *MODULE









                                                                      Bottom
 Command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 124. Work with Module List

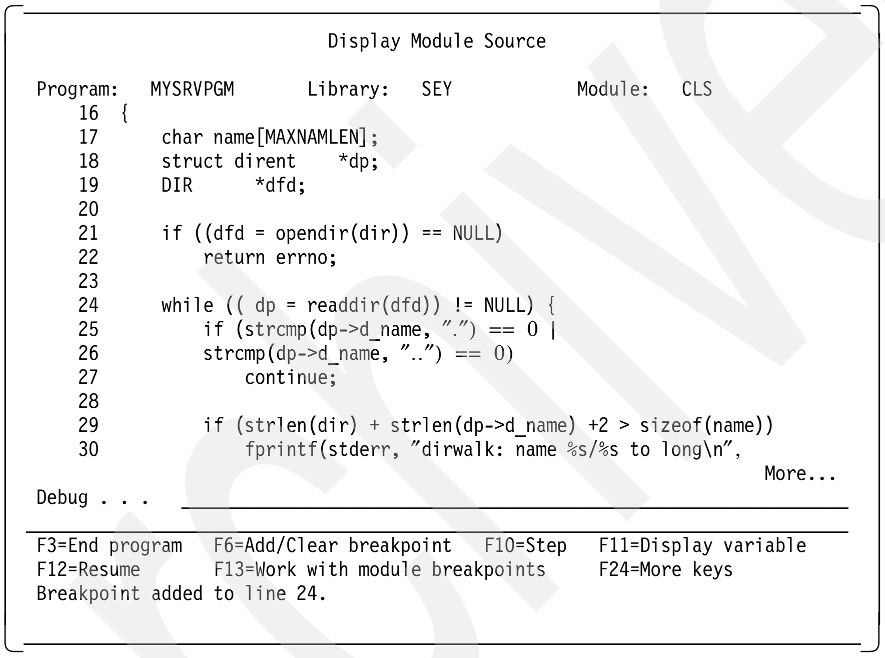
The Figure 125 shows the program/module list after you added the service program.



```
                            Work with Module List

         System:   RCHASM03

          Type options, press enter.

             1=Add program   4=Remove program   5=Display module source

                8=Work with module breakpoints

 Opt     Program/module     Library       Type
 _         MYSRVPGM         *LIBL         *SRVPGM
 _         MYSRVPGM         MYLIB         *SRVPGM
 _           CCD                          *MODULE
 _           CPWD                         *MODULE
 _           CCAT                         *MODULE
 5           CLS                          *MODULE
 _         CPROG            MYLIB         *PGM
 _           CMAIN                        *MODULE     Selected
 _           CINPUT                       *MODULE



                                                                         Bottom
 Command
 ===> ____________________________________________________________________________
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
Program MYSRVPGM added to source debugger.
```

Figure 125. Debug Module List

On the preceding display, when you entered **option 5** on ′CLS′ module, then Display Module Source window for ′CLS′ module is displayed. The Figure 126 shows the source of ′CLS′. To set the break point, move the cursor on the line where you want to set it and press the <F6> key. If the break point is set on the line 26, the line number is highlighted.



```
                             Display Module Source

 Program:   MYSRVPGM       Library:   SEY           Module:   CLS
     16  {
     17      char name[MAXNAMLEN];
     18      struct dirent    *dp;
     19      DIR      *dfd;
     20
     21      if ((dfd = opendir(dir)) == NULL)
     22          return errno;
     23
     24      while (( dp = readdir(dfd)) != NULL) {
     25          if (strcmp(dp->d_name, ″.″) == 0 |
     26          strcmp(dp->d_name, ″..″) == 0)
     27              continue;
     28
     29          if (strlen(dir) + strlen(dp->d_name) +2 > sizeof(name))
     30              fprintf(stderr, ″dirwalk: name %s/%s to long\n″,
                                                                   More...
 Debug . . .   ________________________________________________________________

 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F13=Work with module breakpoints     F24=More keys
 Breakpoint added to line 24.
```

Figure 126. Display Module Source

To debug the program, press the <F12> key, and enter the previous command to run the program. In Figure 126, when you enter **option 1** to display the directory list, the program is stopped at the break point. With the <F10> key, you can run the program step-by-step. You can see where the bug is. In the **if** statement, the directory name is checked whether it is ″.″ or ″..″. A logical OR operation (||) should be used instead of bit OR operation (|).

When you press the <F3> key, you can exit the debugging window. However, the debug session is still active; if you want to end debugging, then you enter the command **ENDDBG** on the command line.

C.5.8 Fixing Errors

To apply the changed ′CLS′ source code, you create module ′CLS′ and service program ′MYSRVPGM′ again. You do not need to create ′CPROG′ again.

List of Abbreviations

AIX	Advanced Interactive eXecutive
API	Application Program Interface
APPN	Advanced Peer-to-Peer Network
ASCII	American National Standard Code for Information Interchange
CISC	Complex Instruction Set Computer
CL	Control Language
CLP	Control Language Program
DCE	Distributed Computing Environment
EBCDIC	Extended Binary Coded Decimal Interchange Code
FTP	File Transfer Protocol
IBM	International Business Machines Corporation
IETF	Internet Engineering Task Force
IFS	Integrated File System
IMPI	Internal MicroProgrammed Interface
IPX	Internetwork Packet eXchange
ISO	International Standards Organization
ITSO	International Technical Support Organization
LAN	Local Area Network
LIC	Licensed Internal Code
MI	Machine Interface
NetBIOS	Network Basic Input Output System
NNTP	Network News Transfer Protocl
OSF	Open Software Foundation
OSI	Open Systems Interconnection
POSIX	Portable Operating System Interface
PROFS	Professional Office System
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SNA	Systems Network Architecture
SNMP	Simple Network Management Protocol
SPX	Sequenced Packet eXchange
TCP/IP	Transmission Control Protocol/Internet Protocol
TIMI	Technology Independent Machine Interface
UUCP	UNIX-to-UNIX Copy Program

Index

Printed in U.S.A.

SG24-4438-00