SG24-4938-00

UNIX C Applications Porting to AS/400 Companion Guide

November 1996

IBM

International Technical Support Organization
Rochester Center

SG24-4938-00

International Technical Support Organization

UNIX C Applications Porting to AS/400 Companion Guide

November 1996

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix F, “Special Notices” on page 115.

First Edition (November 1996)

This edition applies to V3R6 ILE C/400, 5716-CX4, for use with V3R6 OS/400, 5716-SS1.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

Contents

Figures

Tables

Preface

This document is a collection of the knowledge acquired by consultants and IBM experts who have been working with customers porting applications from UNIX system to the AS/400 system.

This document supplements the ITSO Rochester Center redbook, *UNIX C Application Porting to AS/400*, SG24-4438-00, and provides porting information not covered in the previous redbook, such as data conversion, networking, display handling, and so on. It also contains UNIX platform-specific information that you can use when you port C applications to the AS/400 system.

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

To utilize this document effectively, you should have a working knowledge of the AS/400 system and UNIX system and a knowledge of UNIX C and ILE C/400 application development.

How This Redbook Is Organized

This redbook contains 156 pages. It is organized as follows:

- Chapter 1, "Introduction"

  This chapter describes the general purpose of the book. It talks about common portability issues and discusses the two systems (AS/400 and UNIX System) in this perspective.

- Chapter 2, "Overview of Source and Target Environment"

  This chapter discusses the C language environment in the two systems, highlighting their differences. It also mentions the software development tools provided by the AS/400 system that make application development easier.

- Chapter 3, "File Handling"

  This chapter talks about the differences in file handling features in the two systems.

- Chapter 4, "Networking"

  This chapter points out the differences in the TCP/IP features in the source and target systems. It also talks about the APPC implementation differences in the two systems wherever applicable.

- Chapter 5, “Data Conversion”

  This chapter provides a discussion on the Data Conversion APIs of OS/400. Examples are provided illustrating their use.

- Chapter 6, “MI Instruction Function Calls”

  This chapter discusses the MI instruction calls in OS/400 and illustrates how unsupported features of UNIX may be implemented using them.

- Chapter 7, “Message and Error Handling”

  This chapter provides a comparison between the message and error handling techniques in the two systems. Tips on porting these features are also provided with examples.

- Chapter 8, “Display Handling”

  This chapter discusses how a UNIX application can have the user interface redesigned for an AS/400 implementation by using Dynamic Screen Manager (DSM) and other native AS/400 constructs to create menus, panels, and so on.

Appendixes of this document include:

- Appendix A, “HP C to ILE C/400 Application Porting”

  This appendix contains a discussion of the platform-specific features of C on the HP-UX environment and techniques to implement these in ILE C/400.

- Appendix B, “SCO System C to ILE C/400 Application Porting”

  This appendix contains a discussion of the platform-specific features of C on the SCO UNIX environment and techniques to implement these in ILE C/400.

- Appendix C, “Sun Solaris C to ILE C/400 Application Porting”

  This appendix contains a discussion of the platform-specific features of C on the SUN Solaris environment and techniques to implement these in ILE C/400.

- Appendix D, “AIX C to ILE C/400 Application Porting”

  This appendix contains a discussion of the platform-specific features of C on the AIX environment and techniques to implement these in ILE C/400.

- Appendix E, “DEC ALPHA C to ILE C/400 Application Porting”

  This appendix contains a discussion of the platform-specific features of C on the DEC Alpha environment and techniques to implement these in ILE C/400.

A glossary of the new terms introduced in the document is also provided.

The Team That Wrote This Redbook

This redbook was produced by the AS/400 Partners In Development UNIX-to-AS/400 Porting Team in conjunction with Tata Consultancy Services (TCS), Noida, India with the help of the International Technical Support Organization Rochester Center.

Charlie Quigg is a staff programmer in the AS/400 Partners in Development organization. He leads a team as a technical consultant for UNIX-to-AS/400 porting. He coauthored the redbook ″UNIX C Applications Porting to AS/400″ and previously worked in the Rochester Development Lab integrating UNIX-type system interfaces into OS/400.

Rajeev Jain is a Senior Systems Analyst in Tata Consultancy Services, India. He has been working at TCS for 6 years on various assignments including application development, conversion, and maintenance. Currently he is a member of the UNIX-to-AS/400 Porting Team in Partners In Development at IBM Rochester. His area of expertise includes application development on the AS/400 system, UNIX, and Oracle.

Praveen Kumar is a Senior Systems Analyst in Tata Consultancy Services, India. He has been working at TCS for 2 years in the AS/400 field. He has worked at Indian Institute of Technology (IIT), Delhi, India in the field of VLSI. His area of expertise includes the application development on the AS/400 and UNIX system.

Anu Bahri is a Systems Analyst in Tata Consultancy Services. He has been working at TCS for 4 years. His expertise includes application development on the AS/400 system, UNIX, and Oracle.

Rakesh Sarup is an Assistant Systems Analyst in Tata Consultancy Services. He has been working at TCS for 2 years. His area of expertise includes application development on the AS/400 system.

Thanks to the following people for their invaluable contributions to this project:

Jaejin Ahn
ITSO Rochester Center

Rajeev K Arora
TCS, Nodia, India

Comments Welcome

We want our redbooks to be as helpful as possible. Should you have any comments about this or other redbooks, please send us a note at the following address:

redbook@vnet.ibm.com

Your comments are important to us!

Chapter 1. Introduction

1.1 What is Portability?

It is common to speak of portability as a desirable quality for a programming language for code produced by a particular compiler, but what does the term portability really mean? Portability can be divided into two categories:

- **Source code portability:** The ability to take a program that can be compiled, linked, and run on one platform with one compiler and compile, link, and run it one another platform. There are two types of problems associated with source code portability in C:
  - Platform-specific differences: Variations that are a direct result of the characteristics of the platform.
  - Implementation-specific differences: Variations that are a result of behavior that is left unspecified in a given C language definition, and variations that are a result of differences between two distinct C language definitions.
- **Object code portability:** The ability to run compiled and linked object code on more than one platform.

This book deals with source code portability.

1.2 Understanding the Issues and Solutions

This document identifies many differences between the two systems that must be resolved before the porting begins and then constantly monitored throughout the portability process. Once the difference between the two systems is understood, a set of installation standards can be developed for the portability process. As in any programming effort to solve a given programming problem, there can be as many workable solutions as there are programmers. A standard way of handling these differences makes the entire porting process much smoother and ensures that the problems are handled in the same manner by everyone.

This manual does not attempt to discuss the pros and cons of either system but attempts to point out the areas where major differences are found in porting applications written on C from a UNIX system to the ILE C/400 application on an IBM AS/400 system.

1.3 Elements Involved in Application Porting

An application is a collection of programs and data and their interactions with the system environment. Porting an application written in C in a UNIX environment can involve one or more of the following application elements.

- Porting UNIX C Language programs to Integrated Language Environment C/400 programs.
- Porting UNIX command ″shell″ programs to the AS/400 system. This can be in the AS/400 Control Language (CL) or any of the AS/400 languages that can provide an efficient functional equivalent.
- Applications involving specific devices such as communications lines involve the portability of the logical file representations of these devices.
- Porting application data from UNIX to the AS/400 system. This involves flat file porting or database porting from the UNIX to the AS/400 system.

Application portability involves finding efficient functional equivalents wherever possible on the AS/400 system and working around solutions wherever equivalent functions are not available.

1.4 The C Programming Language

The C programming language is available for many software platforms on both IBM and non-IBM equipment. Most language implementations offered on a specific hardware and software environment follow the specifications of the ANSI standard for the C programming language. However, in each environment, the C language often contains extensions to the ANSI standards to better exploit the capabilities offered by the environment. The ILE C/400 language incorporates all of the ANSI language elements and also provides extensions to exploit the AS/400 environment. C languages on UNIX generally follow the X3.159-1989 ANSI C standard. There are many extensions to this ANSI C to exploit its specific architecture in use.

When porting applications from one environment to another, the common elements in a language need little or no change. The language extensions and the extent to which they are used often determine the complexity of the portability project. Other issues depend on using system features in the application.

1.4.1 ILE C/400

The ILE C/400 compiler is a licensed program that has many new features and enhancements, such as:

- Improved run-time performance.

- The compile time for ILE C/400 C programs is less, reducing the amount of time a programmer has to wait for a compilation to complete.
- The C language now has a packed decimal data type allowing the C programs to directly manipulate the packed data in database files.
- Static binding is provided as part of ILE greatly benefits C programs. The programs are statically bound at program creation time rather than only at run time. This reduces the call overhead in calling among modules.
- The *checkout* compiler option tells the compiler to perform further checking of the C program for common programming errors. This helps the programmer create error free programs faster.
- The Dynamic Screen Manager allows the C programmer to control the display from the program dynamically rather than having to create a DDS for display.
- ILE C/400 now includes access to the machine interface (MI).
- It supports the industry standards for C such as ANSI and ISO. This provides greater portability of code from other platforms when written in these standards.

The ILE C/400 compiler improves the run-time performance of applications and helps improve the programmers′ productivity when creating applications.

1.4.2 UNIX C

UNIX C is the generic name for the C language available on UNIX platform of different vendors such as HP, SUN, SCO, and so on. These languages are very similar to each other, but many differences exist among them. Some specific platforms are discussed in the appendixes. Some system and hardware-specific differences exist among machines of different sizes. The language supports ANSI and other standards.

1.5 Portability of Language

Portability is one of the most important characteristics of the C language. However, certain elements of the C language, such as the alignment of members in structures, are highly dependent on the platform on which the C code runs. These elements cannot be standardized for all of the implementations of C. While this book deals with the differences in the C language on the UNIX and AS/400 platforms, most of the elements that make up the definition of C are consistent among the two.

1.5.1 UNIX C

C language on UNIX platforms has been standardized to a great extent but differences are real enough sometimes to hinder porting applications from a UNIX C environment to another UNIX environment. These differences are mainly due to differences in the underlying processors. The C available on a smaller machine is not the same as C available in larger machines. Most versions of C have core features that are ANSI compliant.

1.5.2 ILE C/400

ILE C/400 follows the ANSI standard. However, to exploit system features, many extensions have been provided. These extensions are not supported by C on other environments. Typical extensions are externally described database and distributed database files, display files, ICF (Inter-program Communication Files), extensive natural language support, and so on. Using APIs is another extension that is not available on other systems. ILE C/400 can call unbounded compiled programs of many ILE languages that other C languages do not support. ILE C/400 has more additional types of pointers than UNIX C. ILE C/ 400 also supports Record I/O apart from stream I/O as in UNIX C.

1.6 Expected Portability Differences

Differences are likely to occur when porting UNIX C applications to ILE C/400 because of the following characteristics of programming:

- Differences in the operating system environment
- Unstructured programming
- Using extended compiler options
- Using assembly language format
- Floating point operations
- Data alignment differences
- Data file incompatibility and Input/Output operation
- Memory and address use
  - Using shared memory
  - Absolute addressing
  - Memory organization
  - Architecture dependent memory addressing
- Language differences
  - Using extended language
  - Features based on specific architecture
  - Different semantics of language

Chapter 2. Overview of Source and Target Environment

This chapter describes the differences between the C language constructs on the two systems. The C language constructs such as character sets, data types, data alignment, predefined macros, and preprocessor directives are covered. Other features such as inter-language calling, error handling, and signal handling are covered. Each section contains comments on how to proceed with the conversion when the differences are present.

2.1 Character Encoding

The encoding of characters is not the same on all platforms. The UNIX C products run on platforms that use ASCII encoding, while the AS/400 products run on platforms that use EBCDIC encoding.

If a character value is to be indicated, the character itself should be used rather than the encoding of the character. For example, the code shown in the following example gives a different output in ILE C/400:

```
# include <stdio.h>
main () {
   char x;
   x = ′\x4E′;
   printf(″Here is the character: %c \n″, x):
}
```

The output in UNIX C is:

```
Here is the character: N
```

The output in ILE C/400 is:

```
Here is the character: +
```

The fourth line should be changed to:

```
   x = ′N′ ;
```

2.2 Collating Sequence

Each encoding produces a collating sequence that specifies the order of characters in the encoding. The collating sequences in the UNIX C products are different from the collating sequence in the ILE C/400 products as the UNIX C products run on platforms that use ASCII encoding, while the AS/400 products run on platforms that use EBCDIC encoding.

Portable code should not depend on the relative position of characters in the collating sequence. In particular, portable code should not depend on:

- The contiguity of the letters.
- The relative position of the uppercase and lowercase letters.
- The relative position of the letters and the digits.

If two strings are to be compared (for example, in the strcmp() function), and the differences between collating sequences are to be avoided, these three steps should be followed:

- Create a *locale* that includes a standard collating sequence.
- Use the *setlocale* function to set the locale to the one that was created.
- Compare the strings using the *strcoll* function.

2.3 Character Set

The following lists the basic character set that must be available at both compile and run time:

- The uppercase and lowercase letters of the English alphabet:

  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- The decimal digits 0 to 9:

  0 1 2 3 4 5 6 7 8 9

- The following graphic characters:

  ! ″ # % & ′ () * + , - . / :
  ; < = > ? & [\ ] _ { }

- The *caret* (^) character in ASCII (bitwise exclusive OR symbol) is represented by or the equivalent not (-) character in EBCDIC.
- The *split* vertical bar (|) character in ASCII that may be represented by the *vertical bar* (|) character in EBCDIC systems.
- The space characters.
- The control characters representing horizontal tab, vertical tab, form feed, and end of string.

Uppercase and lowercase letters are treated as distinct characters. If a lowercase is specified as part of an identifier name, one cannot substitute an uppercase in its place. The lowercase letter must be used.

For the keyboards that do not support the entire character set, one can use trigraphs as alternate symbols to represent some characters.

2.4 Trigraphs

2.4.1 UNIX C

UNIX C uses the ASCII character set and all of the C language characters are available on UNIX keyboards. However, reduced character sets that can also be used as **trigraphs** are defined. This trigraph definition is the same as in **ILE C/400.**

2.4.2 ILE C/400

The AS/400 system uses the EBCDIC character set. Some characters used in the C language are not available on some **IBM** keyboards. Sequences of three characters called **trigraphs,** as defined by the ANSI standard, are used to emulate such characters.

The following trigraph sequences may be used in an ILE C/400 source program.

Table 1. Trigraph Sequences in ILE C/400

Trigraph	Character	Name
??=	#	Number sign
??(	[	Left bracket
??)	]	Right bracket
??.	{	Left brace
??>	}	Right brace
??/	\	Back slash
??′	^	Caret
??!	|	Vertical bar
??-	~	Tilde

In the C/400 language, the ^ character in ASCII for the bitwise exclusive OR is represented by the ¬ character in the EBCDIC.

2.4.3 Comments

The square brackets ([and]) can be used on the PC after modifying the keyboard profile for the equivalent EBCDIC hex values. The session profile should be changed to display the square brackets.

2.5 Data Types

The sizes of the C data types vary among the C compilers on different machines. Unlike many other high-level languages that offer a wide variety of basic data types, C offers only a few, keeping the language simple and compact. The basic types are *char,* int, float, and double; a short list indeed, considering that even a character string is not a basic type.

Table 2. Data Types in UNIX C and ILE C/400

Type	**UNIX C** **Size** **(In bytes)**	**ILE C/400** **Size** **(In bytes)**
Char	1	1
Short	2	2
Int	4	4
Long	4	4
Float	4	4
Double	8	8
Long double	16	8
Enum	4	1, 2, or 4
Pointer	4	16

2.5.1 Comments:

- ILE C has a 16-bytes long pointer that is different from the size of int.
- Double float has different sizes on different series of UNIX machines.

2.5.2 Packed Data Type

Packed decimal is a new arithmetic data type that is an extension to ANSI C. It allows representation of a larger number of significant digits (32) useful in business and commercial applications. In ILE C/400, packed decimal has been defined as a separate data type. The include file is `<decimal.h>`. The packed decimal data type is declared, and normal operators (such as arithmetic operators, relational operators, comma operators, assignment operators, conditional operators, equality operators, logical operators, primary operators, and unary operators) work on these. Bitwise operators do not work on these. Packed decimal can be passed as an argument to functions and library functions.

2.6 Data Alignment

According to ANSI (X3.159-1989), alignment is the requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address. An object that has alignment n (where n is a non-negative integer) begins on an address that is divisible by n.

2.6.1 Fundamental Data Types

2.6.1.1 UNIX C

In UNIX C, the data is aligned a little differently on different machines. The alignment of data is:

Table 3. Data Alignment in UNIX C

Type	**Alignment (in bytes)**
Char	1
Short	2
Int	4
Long	4
Float	4
Double	8
Long double (ANSI mode only)	8
Enum	1, 2, or 4
Pointer	4

These default alignments can be changed by using storage modifiers that ensure a particular alignment scheme.

2.6.1.2 ILE C/400

All data types in the ILE C/400 language align on their natural boundary. For example, the **int** fields in a structure align on a four-byte boundary.

Table 4 (Page 1 of 2). Data Alignment in ILE C/400

Type	**Alignment (in bytes)**
Char	1

Table 4 (Page 2 of 2). Data Alignment in ILE C/400

Type	**Alignment (in bytes)**
Short	2
Int	4
Long	4
Float	4
Double	8
Long double (ANSI mode only)	8
Enum	1, 2, or 4
Pointer	16

The following example shows the effect of the alignment:

```
struct {
      char x;
      int y;
      char *p;
} structm;
```

The storage layout for this structure is as follows:

Table 5. Storage Layout of Structure structm on AS/400

Offset	**Bytes Length**	**Field**
0	1	x
1	3	padding
4	4	y
8	8	padding
16	16	p

From the preceding example, clearly a padding of three bytes is used in the structure layout before the integer y, and the padding of eight bytes is used before the pointer p.

The rules that the ILE C/400 compiler and the library follow for storage layout are:

- Padding is added to allow each field to align on its natural boundary.

- Padding is added before an embedded structure to allow the most restrictive field in the embedded structure to align on its natural boundary.

2.6.2 Abstract Data Types - Structures and Unions

Each member in a structure is aligned along a particular address boundary depending on the type of the member. This alignment can leave unused space between members in a structure. This unused space is called padding.

The instruction set and the word size of a system determine, in part, how an element of a particular type can be used efficiently. For this reason, the alignment of a member within a structure depends on the type of member, and the alignment of members of a particular type varies from system to system. So structures, unions, and enumerators suffer most from a data alignment problem. UNIX supports many types of alignments that can be used by pragma directives to the compiler.

The ILE C/400 language provides the **_Packed** storage class to reduce the padding in the structure storage layout. When the **_Packed** is used in a structure declaration, the fields of the structure align on the byte boundary. However, the **ILE** C/400 language pointers are exceptions that always align on a 16-byte boundary.

The preceding structure is declared next with the **_Packed** construct as follows:

```
_Packed struct{
          char x;
          int y;
          char *p;
          } structm;
```

Table 6. Storage Layout of Packed Structure structm on AS/400

Offset	**Bytes Length**	**Field**
0	1	x
1	4	y
5	11	padding
16	16	p

From the preceding example, clearly a padding of 11 bytes is used in the structure layout before the pointer p. The padding of 3 bytes before the integer y is removed. The pointers are exceptions that always align on the 16-byte boundary.

2.7 Pointer Usage

This section describes some differences between ILE C/400 and UNIX C pointers.

2.7.1 UNIX C

In UNIX C, the address pointers are four bytes long. An address can be assigned to an integer, an integer to a pointer, and the address of an object of one type to a pointer to another type. Such assignments are simple copy operations with no conversion.

2.7.2 ILE C/400

In ILE C/400 language, a pointer is an AS/400 system space pointer that points to a system space object. The use of a pointer variable on the AS/400 system is controlled by the hardware and is different from other C platforms. The AS/400 system uses 16-byte pointers. The pointer arithmetic works the same way in the ILE C/400 language as long as the system is aware that this variable is an address pointer.

Programming habits, such as manipulating pointers as integers, does not work on this system. For example:

```
int *p, i, j = 1;
p = &j;          /* initialize p to address of j */
i = (int)p;      /* gives offset into space      */
p = (int*)i;     /* always NULL                  */
```

In statement three, the pointer p is cast to an integer. In statement four, it is cast back to a pointer. Since the compiler does not have enough information to know which space the offset is in, the pointer is set to NULL.

Packed and non-packed structures have different memory layouts in the AS/400 system. Therefore, comparisons and assignments between pointers to packed or non-packed objects may produce an undesirable result.

ANSI C pointers are derived from function type, data object type, or an incomplete type. However, AS/400 pointers can also come from other AS/400 entities, such as system objects, code labels, and process objects. Such pointers do not support the usual pointer operations.

Table 7. Pointer Types on AS/400

Pointer Type	**Function**
Open	Void pointers
Space	Data object pointer
Function	Function pointer
System	Pointer to system objects
Label	Pointers to fixed locations
Invocation	Pointers to space objects
Suspend	Enters to location in a procedure where control has been suspended

A NULL pointer cannot be used in a relational operation with any pointer type. Relational and arithmetic operations are valid only for an OPEN pointer having an address of SPACE pointer or for SPACE pointers.

Rules of pointer casting:

1. When int is cast to a pointer, the result is always a NULL pointer.
2. When an OPEN pointer is cast to int, if it has the address of SPACE pointer at that time, then an offset is returned, otherwise a run-time exception occurs.
3. All other pointers cast to int always give zero.
4. A SPACE pointer cast to int always gives the offset; a NULL pointer cast to int always gives zero.

2.7.3 Comments:

Since a pointer in the ILE C/400 is 16 bytes long, it cannot be treated as an integer as previously shown. A pointer can only be assigned to any other pointer type with proper casting. An assignment to a pointer as in statement four on 12 is not useful. Such statements need to be replaced by some other equivalent statements. In the preceding example, statement four on 12 can be removed.

In a UNIX C program, an integer can be assigned to an address. An address of an object of one type can be assigned to a pointer of another type without a proper pointer casting operation. Such assignment operations may produce undesirable exceptions in an ILE C/400 language program.

Pointers to packed and non-packed objects are quite different. So a packed structure or a packed union cannot a reference non-packed structure or union. Proper type casting is needed in such cases.

2.8 Main() Function

2.8.1 UNIX C

When called from the command line, standard arguments referred to as *argc, argv,* and *envp* (pointer to the environment) can be passed.

2.8.2 ILE C/400

When called from the command line, the arguments *argc* and *argv* can be passed.

2.9 Software Development Tools in AS/400 System

Many tools are available on the AS/400 system. These tools make application development on the AS/400 system easy, consistent, and cost effective. Some of the tools are:

SEU	The source entry editor (SEU) provides editing, syntax checking, and intelligent prompting for application development. However, there is no syntax checking and prompting for free format languages such as C.
SDA	Screen Design Aid (SDA) makes the display and menu design for applications easy and consistent.
PDM	Programming Development Manager (PDM) provides a menu-driven environment to develop application programs.
DFU	Data File Utility (DFU) is the utility to manipulate data in the database files.
IDDU	Interactive Data Definition Utility (IDDU) is used for interactively describing external data.

Other utilities are for character generation (CGU), report layout making (RLU), Advance Function Printer Utility (AFPU), SQLC, and so on.

There are two tools to assist programmers in AS/400 programming. The TMKMAKE tool, an AS/400 version of MAKE, does the repetitive tasks in compiling and binding applications. The CHECKOUT compiler option finds the possible programming errors during compilations.

Note: TMKMAKE is provided ″AS IS″ in the QUSRTOOL library.

2.10 Database Management

The AS/400 system has DB2 for AS/400 Relational Database Management System (RDBMS) integrated into its operating system right from the introduction of it. However, the UNIX system has no RDBMS with it and the RDBMS is provided by the third party. As there is no unique RDBMS associated with UNIX, a porting strategy can be formulated only when the RDBMS of the source is known. However, a few observations are:

- Most prominent DB vendors use SQL as the language for DB operations. The AS/400 system fully supports the ANSI standard SQL. The porting process has to look for extensions of ANSI SQL being used by the DB.
- The AS/400 system supports stored procedure, referential integrity, Triggers, and so on. However, the exact syntax may vary from some of the third party RDBMS. Most of the third party RDBMS allow only SQL in a stored procedure together with some constructs for the control of flow. On the AS/400 system, stored procedures and triggers are written in any of the HLL (COBOL, RPG, C, and so on) that may or may not have embedded SQL.

Chapter 3. File Handling

This chapter describes the differences in the file handling features on the two systems. The features such as file specifications, file organization, record formats, and the function **fopen** are covered. The alternative method of the UNIX type of file system, called Integrated File System, is also discussed.

3.1 File Specifications

3.1.1 UNIX

The hierarchical file system in UNIX is a tree structure with a root denoted by ″/″. A **file** is a node in the tree containing source programs or data, whereas a **directory** contains names and addresses of other files and directories. A file or directory may be accessed by its full or relative path name.

3.1.2 OS/400

The Integrated File System on the AS/400 system supports the hierarchical file system as in UNIX. In addition, it also has its native file system, wherein all data is stored in **objects.** A **library** is an object that contains other objects called **files.** A file has **members** containing program sources or data.

3.2 File Organization and Record Format

3.2.1 UNIX

UNIX treats a file as a continuous **stream** of characters. All I/O operations are carried out using the stream.

3.2.2 OS/400

The AS/400 system also supports stream I/O as in UNIX. In addition, in the native file system of the AS/400 system, files may be made up of records, and I/O operations at the OS level are carried out on records using data management operations. The following are the different types of files in the native file system:

Table 8. File Types on AS/400

File Type	**Function**
Database files	To store data on the AS/400 system.
Device files	To provide access to externally attached devices such as Display, printer, tape, diskette, Intersystem Communications Functions (ICF).
Save files	To save data in a format used for backup and recovery purposes.
Distributed Data Management (DDM) files	To access data on remote systems.

ILE C/400 provides functions for its hierarchical file system as well as its native object based file system. The low-level calls such as *open(), close(), read(),* and so on work on stream files, whereas calls such as *fopen(), fclose (), fread(),* and so on work on database files. However, the latter set can also be made to work on the stream files in QOpenSys by specifying *IFSIO on the *″System Interface Option″* provided on the CRTCMOD command. Besides these, there are APIs such as *_Ropen(), _Rread(), _Rformat(),* and so on for record-oriented file I/O.

Many functions are provided in ILE C/400 that allow file I/O at the record level to improve performance. Many options such as blocking of records, record format, character set id, commitment control, sequence of arrival, indicators, and so on can be exercised.

The *fseek()* function in the ILE C/400 run-time library is functionally different from the same in the UNIX C run-time library. Only the ANSI C elements are supported by the **fseek** function in ILE C/400. Also, the *lseek()* function used in C should be replaced by an equivalent code when porting to the AS/400 system.

Filenames in an ILE C/400 program can be any of the following forms:

Filename
Filename (member-name)
Library-name/filename
Library-name/filename (member-name)
Member-name.filename
Library-name/filename/member-name

The last two forms of the file specifications can be used only in a #include directive.

All characters specified for library-name, filename, or member-name are folded to uppercase unless surrounded by a back slash and quotation mark.

For information on standard C file I/O APIs, see the *ILE C/400 Programmer's Reference.* For information on Integrated File System (POSIX) file I/O APIs, see the *System API Reference.*

Chapter 4. Networking

This chapter discusses the implementation differences of TCP/IP protocols on UNIX and OS/400 with respect to command, command options, and sub commands. Implementation of APPC is also discussed.

4.1 TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) refers to a family of non-proprietary network protocols, of which TCP, providing host-to-host transmission, and IP, providing data routing from source to destination, are two important parts. It consists of a layered structure of protocols that range from low-level, hardware-dependent programs to high-level applications. FTP and TELNET are examples of high-level protocols. One strength of the TCP/IP protocol suite is the availability of these standard applications for a variety of operating environments. The AS/400 system puts TCP/IP into effect as part of OS/400 and as part of a licensed program called TCP/IP Connectivity Utilities/400.

4.1.1 Application Protocol Standards

The TCP/IP Connectivity Utilities/400 licensed program contains the following high-level applications and their associated CL commands:

- File Transfer Protocol (FTP)
- TELNET Protocol (TELNET)
- Simple Mail Transfer Protocol (SMTP)
- Remote Printing (LPR and LPD)
- Simple Network Management Protocol (SNMP)
- TCP/IP File Server Support/400

The following sections compare the options and sub-commands of the major applications with the corresponding UNIX implementation. For further details on the AS/400 implementation, see the *OS/400 TCP/IP Configuration and Reference.*

4.1.2 FTP

FTP allows you to transfer data between local and remote hosts. OS/400 TCP/IP supports both client and server FTP functions. The FTP server on OS/400 fully supports the sub commands provided by ***ftpd,*** the UNIX FTP server process. The inactivity time-out period that is passed as a parameter to ftpd, can be changed by the CHGFTPA command on the AS/400 system.

To change the inactivity time-out period, the AS/400 system also supports the FTP server TIME subcommand, which can be sent to the server with the FTP client QUOTE subcommand.

The FTP client on the AS/400 system supports almost all of the sub commands provided by its UNIX counterpart except for certain enhanced features such as defining macros, proxy execution of commands on a secondary control connection, filename mapping/ filename character translation mechanism, storing files on the local system with unique file names. The FTP client on the AS/400 system cannot send files located in the Root, QOpenSys, and QLanSrv file systems.

4.1.3 TELNET

The TELNET protocol allows the client to access and to use the resources of a remote system (the server) as if the client were locally connected to the remote system. AS/400 TCP/IP TELNET provides client and server support that allows a remote logon to hosts within an internet. The AS/400 TELNET client allows an AS/400 TCP/IP user to sign on to and use applications on a remote system that has a TELNET server application.

The AS/400 TELNET client provides control functions that allow you to control workstation processing on the server system when you are in a TELNET session. These correspond to the **send** command of the UNIX TELNET client (Telnet). The AS/400 TELNET client can send IP(Interrupt), AYT(Are You There), and QUIT sequences but does not provide for sending ESCAPE, BRK(Break), EC(Erase Character) and EL(Erase Line) sequences. The UNIX counterpart provides other commands to:

- Open and close connection to hosts.
- Suspend Telnet.
- Set and display Telnet variables (escape, erase, and so on) and boolean flags.

These are not supported by the OS/400 TELNET client. The AS/400 system allows the use of System Request Key and Print key functions during a Telnet session.

4.1.4 SMTP

The Simple Mail Transfer Protocol (SMTP) function allows you to send or receive electronic mail. For consistency with other AS/400 mail functions, SMTP is coupled to the AS/400 SNA distribution services (SNADS). SNADS is part of OS/400 and it contains extensions to support SMTP. SNADS allows you to send mail to various types of users (not just SMTP users) with one consistent user interface.

4.1.5 Remote Printing (LPR and LPD)

It is possible in a TCP/IP network to send your spooled files to any system in your network. The term often used by UNIX TCP/IP software to describe this support is line printer requester (LPR). LPR is the sending, or client portion, of a spooled file transfer. On the AS/400 system, the Send TCP/IP Spooled File (SNDTCPSPLF) command provides this function by allowing you to specify what system you want the spooled file printed on and how you want it printed.

The printing of the file is done by the printing facilities of the destination system. On the AS/400 system, the line printer daemon (LPD) is the process on the destination system that receives the file sent by the SNDTCPSPLF command.

4.1.6 TCP/IP File Server Support/400

The TCP/IP File Server Support /400 allows you to manipulate files on remote TCP/IP hosts as if they reside on your local host. It is based on NFS protocol and uses RPC (Remote Procedure Call) protocol to communicate between the client and the server. TCP/IP File Server Support/400 is a server-only implementation of NFS.

4.1.7 Application Program Interface (API) To TCP/IP

The application program interface (API) to the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) layers on the AS/400 system enables you to write programs that communicate with other systems using TCP/IP.

Many times, an enterprise has unique interoperability requirements for its private networks. This means that the enterprise must provide its own applications to fulfill these unique requirements. On the AS/400 system, this is accomplished with several programming interfaces. These are:

- Sockets Interface
- Pascal API

A socket interface (sockets) allows you to write your own applications to supplement those supplied by TCP/IP. Sockets allows unrelated processes to exchange data locally and over networks. Both connection-oriented and connectionless communications are provided for TCP/IP. With this support, you can write applications to the TCP, UDP, and IP protocols directly. The TCP/IP applications that run on sockets are FTP, SMTP, SNMP, LPR, and LPD. The sockets interface operates over TCP/IP or (by using AnyNet/400) over SNA.

The TCP/UDP programming interface provides a system programmer with a programming interface to TCP or UDP as a set of procedure calls from an AS/400 Pascal program. The TELNET application and Network File Server support use the Pascal API and can only run on TCP/IP.

4.2 Sockets

In this section, we discuss the sockets interface as implemented in the AS/400 system and compare it with the BSD (Berkeley Software Distributions) implementation. For details of the AS/400 sockets implementation, see the *OS/400 Sockets Programming.*

The sockets application programming interface (API) is available only from Integrated Language Environment (ILE) C/400 language programs. It is the standard API for Transmission Control Protocol/Internet Protocol(TCP/IP), and supports a UNIX domain and TCP/IP as part of OS/400. For V3R6, OS/400 also supports Novell NetWare.

Sockets is a Berkeley Software Distributions (BSD) interface. The semantics such as the return codes that an application receives and the arguments available on supported functions are BSD semantics. Some BSD semantics, however, are not available in the OS/400 implementation, and changes may need to be made to a typical BSD socket application in order for it to run on AS/400 systems.

The following table summarizes the differences between the OS/400 implementation and the BSD implementation.

Table 9. Comparison of Sockets Implementation in AS/400 and BSD

OS/400 File	**UNIX File**	**Contents**
QATOCHOST	/etc/hosts	List of host names and the corresponding IP addresses
QATOCPP	/etc/protocols	List of protocols used in the Internet
QATOCPS	/etc/services	List of services, specific port, and protocol a service uses
QATOCPN	/etc/networks	List of networks and the corresponding IP addresses

- /etc/resolv.conf

  The OS/400 implementation requires that this information be configured using the CFGTCP menu.

- bind()

On a BSD system, a client can create an AF_UNIX socket using socket(), connect to a server using connect(), and then bind a name to its socket using bind(). The OS/400 implementation does not support this scenario (the bind() fails).

- close()

  The OS/400 implementation supports the linger timer for close() except for AF_INET sockets over SNA. The BSD implementation does not support the linger timer for close().

- connect()

  On a BSD system, if a connect() is issued against a socket that was previously connected to an address and is using a connectionless transport service, and a not valid address or a not valid address length is used, the socket is no longer connected. The OS/400 implementation does not support this scenario (the connect() fails and the socket is still connected).

  A connectionless transport socket for which a connect() has been issued can be disconnected by setting the address_length parameter to zero and issuing another connect().

- ioctl()

  On a BSD system, with a socket of type SOCK_DGRAM, the FIONREAD request returns the length of the data plus the length of the address. On the OS/400 implementation, FIONREAD only returns the length of data.

  Not all requests available on most BSD implementations of ioctl() are available on the OS/400 implementation of ioctl().

- listen()

  On a BSD system, issuing a listen() with the backlog parameter set to a value that is less than zero or greater than {SOMAXCONN} does not result in an error. In addition, the BSD implementation, in some cases, does not use the backlog parameter, or uses some algorithm to come up with a final result for the backlog value. The OS/400 implementation returns an error if the backlog value is not between zero and {SOMAXCONN}, and setting the backlog to a valid value results in the value being used as the backlog.

- OOB data

  In the OS/400 implementation, OOB data is not discarded if SO_OOBINLINE is not set, OOB data has been received, and the user then sets SO_OOBINLINE on. The initial OOB byte is considered normal data.

- protocol parameter of socket()

As a means of providing additional security, no user is allowed to create a SOCK_RAW socket specifying a protocol of IPPROTO_TCP or IPPROTO_UDP.

- res_xlate() and res_close()

  These functions are included in the resolver routines for the OS/400 implementation. The res_xlate() function translates DNS packets from EBCDIC to ASCII and from ASCII to EBCDIC. The res_close() function is used to close a socket that was used by res_send() with the ES_STAYOPEN option set. It also resets the _res structure.

- sendmsg() and recvmsg()

  On the OS/400 implementation, access rights cannot be passed to a descriptor using sendmsg() and recvmsg(). However, a job can pass access rights to a descriptor using givedescriptor() and takedescriptor().

  The OS/400 implementation of sendmsg() and recvmsg() allows up to and including {MSG_MAXIOVLEN} I/O vectors. The BSD implementation allows MSG_MAXIOVLEN - 1} I/O vectors.

- shutdown()

  The OS/400 implementation of shutdown() may block if an output function is currently blocked on the socket descriptor. On a BSD implementation, the blocking output function is ended with the EPIPE errno value. Similarly, a BSD implementation ends blocking input operations with a zero output value when they are blocking and a shutdown() is issued from another process or thread. The OS/400 implementation simply fails any subsequent input function with a zero output value, but the blocking input function continues to block until data is received or some other action is taken to remove it from a waiting state.

- SO_REUSEADDR option

  On BSD systems, a connect() on a socket of family AF_INET and type SOCK_DGRAM causes the system to change the address to which the socket is bound to the address of the interface that is used to reach the address specified on the connect(). For example, if you bind a socket of type SOCK_DGRAM to address INADDR_ANY, and then connect it to an address of a.b.c.d, the system changes your socket so it is now bound to the IP address of the interface that was chosen to route packets to address a.b.c.d. In addition, if this IP address that the socket is bound to is a.b.c.e, for example, address a.b.c.e now appears on getsockname() instead of INADDR_ANY, and the SO_REUSEADDR option must be used to bind any other sockets to the same port number with an address of a.b.c.e.

In contrast in this example, the OS/400 implementation does NOT change the local address from INADDR_ANY to a.b.c.e. getsockname(), continues to return ADDR_ANY after the connect is performed, and the SO_REUSEADDR option has no meaning for a socket of type SOCK_DGRAM.

- SO_SNDBUF and SO_RCVBUF options

  The values set for SO_SNDBUF and SO_RCVBUF on a BSD system provide a greater level of control than on an OS/400 implementation. On an OS/400 implementation, these values are taken as advisory values.

4.3 APPC

The AS/400 Advanced Program-to-Program Communications (APPC) support is the AS/400 system implementation of the Systems Network Architecture (SNA) logical unit (LU) type 6.2 and node type 2.1 architectures.

The APPC support handles all of the SNA protocol requirements when the AS/400 system is communicating with a remote system using the LU type 6.2 and node type 2.1 architectures. You can connect your system to any other system that supports the APPC program interface. APPC application programs can also communicate over lines using the Internet Protocol (IP) of Transmission Control Protocol/Internet Protocol (TCP/IP). This is achieved by the Multiprotocol Transport Networking (MPTN) architecture. On the AS/400 system, the MPTN architecture is put into effect as AnyNet that is a family of products that allow applications written for one type of network protocol to be run over a different type of network protocol. The AnyNet support allows APPC application programs (such as ICF or CPI Communications applications) to communicate between systems in a TCP/IP network. The systems running the APPC application programs must both have APPC over TCP/IP support.

The APPC protocol consists of a set of verbs that are common to the local and remote systems in a network. However, the way in which each system provides a program interface to the verbs may differ.

The AS/400 system provides the following program interfaces:

- The Intersystem Communications Function (ICF) file interface
- The Common Programming Interface (CPI) communications call interface
- The CICS file interface
- The sockets application program interface (API)

In ICF, the LU 6.2 verbs are implemented using data description specifications (DDS) keywords and system-supplied formats. If you use ICF, you can write application programs using the following languages:

- ILE C/400
- ILE COBOL/400
- FORTRAN/400
- ILE RPG/400

In CPI Communications, the LU 6.2 verbs are implemented using CPI communications calls. If you use CPI communications, you can write application programs in the following programming languages:

- ILE C/400
- ILE COBOL/400
- FORTRAN/400
- REXX/400
- ILE RPG/400
- Cross System Product (CSP)

In CICS/400 support, the LU 6.2 verbs are implemented using EXEC CICS commands. If you use CICS, you can write application programs using the ILE COBOL/400 language.

For the sockets API, the LU 6.2 verbs are implemented using the socket functions. If you use sockets, you can write application programs using the ILE C/400 language. Both the source and target programs must use the sockets API.

APPC is also implemented on some UNIX environments. For example, AIX provides the CPI communications call interface to APPC. Refer to Appendix D, “AIX C to ILE C/400 Application Porting” on page 97 for a comparison of CPI-C implementation on AIX and the AS/400 system.

Chapter 5. Data Conversion

Most UNIX systems run on hardware that uses the ASCII character set to encode data, whereas the AS/400 system uses the EBCDIC set to encode data. This should not cause problems in porting source code if the application makes no assumptions about the character set. However, the files containing data need to be converted to the correct format according to the requirements. To facilitate this task, the AS/400 system provides data conversion APIs. We discuss them in the following sections.

5.1 Data Conversion APIs

The X/Open, CAE specification defines a set of functions that provide data conversion capabilities from the program level. These require the user to initialize a data conversion stream, identifying both the source and the target data. The AS/400 system uses Coded Character Set Identifier (CCSID) values to identify the source and target data. Once initialized, character sequences encoded in the source CCSID can be converted, and equivalent sequences encoded in the target CCSID are returned.

In this section, we discuss the following data conversion APIs:

1. **QDCXLATE - Convert Data** - converts data through the use of a table object.
2. **iconv() - Code Conversion** - converts a buffer of characters from one CCSID into another.
3. **iconv_close() - Code Conversion Deallocation** - closes the conversion descriptor (cd) that was initialized by the iconv_open() or QtqIconvOpen() function.
4. **iconv_open() - Code Conversion Allocation** - performs the necessary initializations to convert character encoding and returns a conversion descriptor of type iconv_t.
5. **QtqIconvOpen() - Code Conversion Allocation** - performs the necessary initializations to convert character encoding and returns a conversion descriptor. The only difference between this function and the iconv_open() function is the format of the parameters.

For details of the data conversion APIs, see the *AS/400 System API Reference.*

An example illustrating the use of a few of these is included at the end of the chapter.

In general, the conversion process of the data conversion APIs comprises the following three steps:

1. Open a conversion descriptor with a specified CCSID pair (through iconv_open() or QtqIconvOpen()).
2. Do multiple conversions (through iconv()).
3. Close the conversion descriptor when done (through iconv_close()).

The API QDCXLATE converges these three steps into one with slight modifications.

Note: Among the APIs previously mentioned, only *iconv(), iconv_open(),* and *iconv_close()* conform to the X/Open industry standard functions.

5.2 Convert Data (QDCXLATE) API

5.2.1 Required Parameters:

- Length of data being converted
- Conversion data
- Single-Byte Character Set (SBCS) conversion table name

5.2.2 Optional Parameters:

- SBCS conversion table library name
- Output data
- Length of output buffer
- Length of converted data
- Double-Byte Character Set (DBCS) language
- Shift-out and shift-in characters
- Type of conversion

The Convert Data (QDCXLATE) API converts data through the use of a table object. The conversion table that QDCXLATE uses for the conversion may be created by the user, or an IBM-supplied table may be used from the QUSRSYS library. When the QDCXLATE API is called with the mandatory parameters only, it converts single-byte data. When all parameters are specified, DBCS conversion takes place. The QDCXLATE API can distinguish double-byte from single-byte characters when converting from ASCII to EBCDIC and vice-versa if the proper parameters have been supplied. The required conversion table may be created by the user, or any IBM-supplied table may be used.

Return Value: The QDCXLATE API converts data byte for byte and returns the converted data to the program. When only single-byte data is converted, the input (unconverted) data is replaced with the converted data. When double-byte data is converted, the converted data is placed in the output data parameter.

Note: For ASCII to EBCDIC DBCS conversion, the type of conversion parameter is mandatory. The possible values are:

*AE Convert ASCII to EBCDIC
*EA Convert EBCDIC to ASCII

5.3 Code Conversion API - iconv()

5.3.1 Syntax

```
size_t iconv (cd, inbuf, inbytesleft, outbuf, outbytesleft)
iconv_t cd;
char    **inbuf;
size_t  *inbytesleft;
char    **outbuf;
size_t  *outbytesleft;
```

The iconv() function converts a buffer of characters specified by the inbuf parameter from one coded character set identifier (CCSID) into another CCSID and stores the converted characters into a buffer specified by the outbuf parameter. The CCSIDs used are those in the conversion descriptor, cd, which was returned from the call to either the iconv_open() or the QtqIconvOpen() function. On input, the inbytesleft parameter indicates the number of bytes in inbuf to be converted. Similarly, the outbytesleft parameter indicates the number of bytes available in outbuf. These values are decremented when the conversion is done, such that on return, they indicate the state of their associated buffers. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that cause the output buffer to overflow. During conversion, iconv() may encounter valid characters in the input buffer that do not exist in the target CCSID. This is known as a character mismatch. In this case, iconv() performs the conversion based on the conversion alternative specified in the *fromcode* parameter of the iconv_open() function.

Return Value: If the entire input buffer is successfully converted, iconv() may return the number of non-identical conversions performed based on the substitution alternative. Otherwise, a zero is returned. If an error occurs, iconv() returns (size_t)-1, and errno is set to indicate the error.

5.3.2 Error Conditions

The following errors can be returned in errno:

[E2BIG]	Insufficient space. Conversion stopped due to lack of space in the output buffer or there was not enough space to store the NULL character in the output buffer.
[EBADDATA]	Shift state not valid in input data. The beginning shift state of the input data buffer does not correspond to the shift state of the conversion descriptor. A shift-state sequence was encountered that tried to change the shift state of the input buffer to the current shift state of the conversion descriptor. For example, an EBCDIC shift-in control character may have been encountered while the conversion descriptor indicated single-byte state. This error is only supported for EBCDIC mixed-byte (X′1301′) encoding schemes.
[EBADF]	Descriptor not valid. The conversion descriptor (cd) parameter is not valid.
[ECONVERT]	The mixed input data contained DBCS characters. Input conversion stopped due to the occurrence of DBCS characters in the input data when converting from a mixed-byte encoding scheme. The shift state for EBCDIC mixed data remains in the initial single-byte shift state. This error can only be returned when the mixed error option has been set accordingly for the QtqIconvOpen() or iconv_open() function.
[EFAULT]	Bad address. The system detected an address that was not valid when attempting to use an argument from the parameter list. An escape message may also be signaled as a result.
[EINVAL]	Parameter not valid. The conversion stopped because of an incomplete character or shift state sequence at the end of the input buffer.
[ENOBUFS]	Number of bytes for the input or output buffer not valid, or the input length cannot be determined. The specified number of bytes for inbytesleft or outbytesleft is not valid. If the input length option field (on the call to iconv_open() or QtqIconvOpen()) specifies that iconv() determines the length of the input buffer and if iconv() cannot find a NULL character in the input buffer, this error is returned.
[ENOMEM]	Not enough space. Insufficient storage space was available to perform the conversion.

[EUNKNOWN]	Undetected error. An undetected error occurred. Contact your service organization. An escape message may also be signaled as a result.

5.4 Code Conversion Deallocation API - iconv_close()

5.4.1 Syntax

```
int iconv_close (cd)
iconv_t  cd;
```

The iconv_close() function closes the conversion descriptor (cd) that was initialized by the iconv_open() or QtqIconvOpen() function.

Return Value: If an error occurs, iconv_close() returns a value of -1 and errno is set to indicate the error. If iconv_close() completes successfully, a value of zero is returned.

5.4.2 Error Conditions

The following errors can be returned in errno:

[EBADF]	Descriptor not valid. The conversion descriptor (cd) parameter is not valid.
[EUNKNOWN]	Undetected error. An undetected error occurred. Contact your service organization. An escape message may also be signaled as a result.

5.5 Code Conversion Allocation API - iconv_open()

5.5.1 Syntax

```
iconv_t iconv_open (tocode, fromcode)
char *tocode;
char *fromcode;
```

The iconv_open() function performs the necessary initializations to convert character encoding from the source CCSID identified by the fromcode parameter to the CCSID identified by the tocode parameter. It then returns a conversion descriptor of data type iconv_t. This API performs the same function as the QtqIconvOpen() API (described later in the chapter) except that the input types of fromcode and tocode are character strings. The conversion descriptor remains valid in a job until that job closes it with iconv_close() or the job ends. The components of the *fromcode* and the *tocode* parameters are elaborated in the *AS/400 System API Reference.*

Return Value: If successful, iconv_open() returns a conversion descriptor of data type iconv_t. This conversion descriptor must be passed unchanged as an input parameter to the iconv() and iconv_close() functions. If unsuccessful, iconv_open() returns -1 in the return value of the conversion descriptor and sets errno to indicate the error.

5.5.2 Error Conditions

The following errors can be returned in errno:

[EFAULT]	Bad address. The system detected an address that was not valid when attempting to use an argument from the parameter list. An escape message may also be signaled as a result.
[EINVAL]	Parameter not valid. The conversion specified in the fromcode and tocode parameters is not supported. When an errno value of EINVAL is returned, check the fromcode and tocode parameters for CCSIDs that are not valid or unsupported alternatives and options.
[ENOMEM]	Not enough space. Insufficient storage space is available.
[EUNKNOWN]	Undetected error. An undetected error occurred. Contact your service organization. An escape message may also be signaled as a result.

5.6 QtqIconvOpen() - Code Conversion Allocation API

5.6.1 Syntax

```
iconv_t QtqIconvOpen (tocode, fromcode)
         QtqCode_T     *tocode;
         QtqCode_T     *fromcode;
```

The QtqIconvOpen() function performs the necessary initializations to convert character encoding from the source CCSID identified by the fromcode to the CCSID identified by the tocode. It then returns a conversion descriptor of data type iconv_t. This API performs the same function as the iconv_open() API except that the input type of fromcode and tocode is of data type QtqCode_T. The conversion descriptor remains valid in a job until that job closes with iconv_close() or the job ends.

Return Value: If successful, QtqIconvOpen() returns a conversion descriptor of data type iconv_t. This conversion descriptor must be passed unchanged as an input parameter to the iconv() and iconv_close() functions. If unsuccessful, QtqIconvOpen() returns -1 and in the return value of the conversion descriptor and sets errno to indicate the error.

5.6.2 Error Conditions

The following errors can be returned in errno:

[EFAULT]	Bad address. The system detected an address that was not valid when attempting to use an argument from the parameter list. An escape message may also be signaled as a result.
[EINVAL]	Parameter not valid. The conversion specified in the *fromcode* and *tocode* parameters is not supported. When an errno value of EINVAL is returned, check the *fromcode* and *tocode* parameters for CCSIDs that are not valid or unsupported alternatives and options.
[ENOMEM]	Not enough space. Insufficient storage space is available.
[EUNKNOWN]	Undetected error. An undetected error occurred. Contact your service organization. An escape message may also be signaled as a result.

The following examples illustrate the fundamental steps involved in using the functions *iconv_open(), iconv()* and *iconv_close()* for ASCII to EBCDIC SBCS data conversion. Figure 1 on page 36 demonstrates the use of the API functions, while Figure 2 on page 37 shows how the same objective can be accomplished using the QDCXLATE API.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <iconv.h>              /*      contains the Data Conversion APIs           */
#include <errno.h>

   extern iconv_t iconv_open (char *, char *);
   extern size_t   iconv (iconv_t, char **, size_t *, char **, size_t *);
   extern int      iconv_close (iconv_t);

main(int argc, char *argv[])
{
/*     All parameters named as in the explanation given earlier                     */

iconv_t cd;

/*    The fromcode and tocode parameters are character strings formed               */
/*    by a concatenation of different substrings depending on the                   */
/*    conversion specifications.                                                    */
char fromcode[33]    = ″IBMCCSID003670000101″;
char tocode[33]      = ″IBMCCSID00037″;

/*    inputstr and outputstr would contain the source and the converted             */
/*    strings respectively.                                                         */
char *inputstr = NULL;
char *outputstr = NULL;

char **inbuf = &inputstr;  /*  function iconv() requires pointers                   */
char **outbuf = &outputstr;         /*           of type (char **)                  */

size_t *inbytesleft, *outbytesleft, result;
int count;

memset(&fromcode[20],′/0′, 13);     /*  The remaining positions in                  */

                                    /*  the fromcode and tocode                     */
memset(&tocode.[13],′/0′, 20);      /*  parameters must be set to                   */
                                    /* hexadecimal zeros.                           */
inbytesleft = malloc(sizeof(size_t));
outbytesleft = malloc(sizeof(size_t));
*inbytesleft = 50;
*outbytesleft = 50;
inputstr = (char *) malloc(50 * sizeof(char));
outputstr = (char *) malloc(50 * sizeof(char));

strcpy(inputstr, ″ABCDEFGHIJ″);/* String of characters in ASCII                     */
strcpy(outputstr, ″″);         /* This would contain the converted text in EBCDIC   */
cd = iconv_open (tocode, fromcode);     /* returns a conversion descriptor          */
result = iconv (cd, &inputstr, inbytesleft, &outputstr, outbytesleft);
                                        /* converts data acording to specifications */
iconv_close (cd)                        /*     closes the conversion descriptor     */
};
```

Figure 1. Use of APIs iconv(), iconv_close(), iconv_open() in Converting Data

The same conversion is achieved by using only the API QDCXLATE with the appropriate parameters as shown in the following figure:

```
#include <decimal.h>

#ifdef  __ILEC400__
#pragma linkage(QDCXLATE,OS,nowiden)
#else
extern ″OS″
#endif

void QDCXLATE (_Decimal(5,0) *, char *, char *);

main( )
{
_Decimal(5,0) length = 10;
char inputstr[20] = ″ABCDEFGHIJ″;
char tablename[10] = ″QASCII       ″;

QDCXLATE(&length, inputstr, tablename);
return;
};
```

Figure 2. Use of API QDCXLATE in Converting Data

Chapter 6. MI Instruction Function Calls

MI (machine interface) library functions provide many of the system level functions required by programmers. These functions can be used as basic building blocks to create more functions, if required. For the analyst porting UNIX applications to the AS/400 system, these functions are of immense value as MI instruction function calls help in providing workarounds for unsupported APIs or library functions. This chapter gives an overview of MI functions. An example of designing UNIX type APIs is included.

For details on the machine interface library functions, see the *ILE C/400 Programmer's Reference*.

6.1 MI Instruction Function Calls

MI instruction calls are low-level system interfaces accessible in ILE C/400 that provide system level programming capabilities. Some of the areas where MI instruction calls provide help are:

- Data conversions such as:
  - ASCII to HEX
  - Char to SNA format
  - Left adjusted byte copying with or without padding
  - Compressing or decompressing of data
  - Translation using tables
- Computation and branching
- Date/Time/Timestamp manipulations
- Pointer/Name resolution addressing
- Space object addressing
- Space management
- Program management
- Program execution
- Independent index
- Queue management
- Object lock management
- Authorization management
- Process management
- Resource management
- Machine observation
- Machine interface support
- Mutex
- Job information

6.2 Interfaces for the Machine Interface

The machine interface (MI) is the machine instruction set that allows access to low-level machine procedures. Most of the MI instructions can be accessed through two interfaces: the built-in interface and the function interface. Some of the functions in the ILE C/400 MI library do not have a built-in interface, and some of the built-ins do not have a function interface.

The built-in interface is a built-in routine that directly accesses the low-level machine procedure. The address of a routine through its built-in interface is not available. Also, there is no stack frame associated with a call to a routine through its built-in interface. Performance may be improved if a built-in interface is used. However, the machine procedures do not use a consistent parameter passing mechanism nor do they make use of return values and null-terminated strings such as C library functions.

Although built-in versions are mentioned in the discussion of the function, the syntax is not provided. Refer to the *Machine Interface Functional Reference* where the syntaxes of all the MI built-ins are provided.

The function interface provides an easier, more consistent, way for passing parameters and using the ILE C/400 function calling conventions. If the parameter lists of the function interface and the built-in interface to a machine procedure are identical, the function interface is available as a macro that maps directly to the built-in.

6.3 MI Instruction Header Files

The ILE C/400 MI header files contain definitions for both the built-in and function interface to an MI instruction. In releases prior to V3R6, the ILE C/400 MI functions and built-ins were declared in header files according to their use (MI group header files), located in library QCLE. In V3R6 and following releases, the MI header files are residing in library QSYSINC, file MIH, with member names identical to the name of the header files.

6.4 Sample Workaround Using MI Instruction Calls

One of the APIs of UNIX that is not directly supported on the AS/400 system is *lockf*. This is available in HP-UX for providing semaphore and record locking facilities in the file. API lockf takes the file descriptor, control function, and offset size as argument, and puts a lock on the specified region. The syntax is:

```
int lockf(int fildes, int function, off_t size);
```

fildes provide the file descriptor to the API lockf,

function provides the type of operation to be performed,

size is the number of contiguous bytes to be locked.

Some behavioral characteristics of lockf are:

- The size can take negative arguments. The locking in this case goes in the backward direction until offset size.
- Locking can cross the file boundary, if the size of offset (size) is so.
- Locking is valid until file is open and the process that called lockf exists.

In the following example, these basic characteristics are provided. Some error conditions, contiguity checks, and so on have been simplified to keep the example code simple.

```
/*   This module gives a workaround for UNIX API ″lockf″.  The       */
/*   UNIX API lockf gives file locking/semaphore type locking        */
/*   facility for the specified portion of the file.  The syntax is  */
/*                                                                   */
/*   #include <unistd.h>                                             */
/*   int lockf(int fildes, int function, off_t size)                 */
/*                                                                   */
/*   where:                                                          */
/*                                                                   */
/*   fildes is an open file descriptor,                              */
/*   function is control value that specifies action to be taken     */
/*         F_TEST:  test if the required portion is locked           */
/*         F_LOCK:  lock the required portion,                       */
/*         F_TLOCK: test if lock exist on the required portion,      */
/*              then lock if required.                               */
/*         F_ULOCK: Unlock the required portion                      */
/*                                                                   */
/*   size is the number of contiguous bytes to be                    */
/*   locked/unlocked.                                                */
/*                                                                   */
/*   The example uses MI functions available in the ILE C/400        */
/*   library to give the workaround.  The MI functions used are:     */
/*                                                                   */
/*     matobjlk()                                                    */
/*     unlocksl()                                                    */
/*     locksl()                                                      */
/*                                                                   */
/*   The example shows the workaround for basic functionality.       */
/*   Some details have been omitted for the sake of clarity of       */
/*   the example code.  For instance, size  0 in UNIX locks file     */
/*   until the end of file (EOF).  Here it locks only one byte       */
/*   in the file.  See HP-UX Reference Vol-2, Section 2 for full     */
/*   description of lockf.                                           */
```

Figure 3 (Part 1 of 3). API Lockf()

```
#include <stdlib.h>
#include <recio.h>
#include <stdio.h>
#include <errno.h>
#include <milock.h>
#include <mispcobj.h>
#include <milib.h>
#define F_ULOCK 0
#define F_LOCK 1
#define F_TLOCK 2
#define F_TEST 3

_MOBJL_Template_T    allocated_locks;
_RFILE    *file_pointer;
char     *fp;
int locked=0, result=0, i;
int lockf(_RFILE *fildes, int function, int size) {

/*  The fildes is a pointer to the type _RFILE.  It is converted     */
/*  to pointer of type char to make it address a data type           */
/*  of size 1 char.                                                  */

    fp = (char *) fildes;

/*  If size is negative, decrement the loop index and pointer        */

    if (size < 0) {
       fp += size;
       size = size * -1;
    }

/*  Test if region is already locked for F_TEST & F_TLOCK            */

    if: ((function == F_TEST) || (function == F_TLOCK)) {
        for (i =0; i < size +1; i++, fp++){
            matobjlk(&allocated_locks, fp);

/*  If lenr bit is set                                               */

            if (allocated_locks.Lock_Alloc & _LENR_LOCK) {
                locked += 1;
            }
      }

        if ((function == F_TEST) && (locked > 0)) {
              printf("%d bytes are locked \n", locked);
              return(0);
      }

        if ((function == F_TLOCK) && (locked > 0)){
             printf("%d bytes are already locked \n ", locked);
             return(-1);
      }

   }
```

Figure 3 (Part 2 of 3). API Lockf()

```
/*  F_ULOCK needs unlocking a locked region                         */

  if (function == F_ULOCK) {
     for (i =0 ; i < size +1; i++,fp++){
         matobjlk(&allocated_locks, fp);
         if (allocated_locks.Lock_Alloc & _LENR_LOCK)
             unlocksl(fp, _LENR_LOCK);
         matobjlk(&allocated_locks, fp);

/*  If LENR bit is bit is still set                                 */

            if (allocated_locks.Lock_Alloc & _LENR_LOCK){
                 printf("Error. The LENR lock still exists.\n");
                 exit(-1);
            }
          }
        }
/*  F_LOCK & F_TLOCK need locking an unlocked region                */

  if ((function == F_LOCK) || (function == F_TLOCK)) {
       for (i =0 ; i < size +1; i++,fp++){
            locksl(fp, _LENR_LOCK);
            matobjlk(&allocated_locks, fp);

/*  If lenr bit is set                                              */

       if (allocated_locks.Lock_Alloc & _LENR_LOCK) {
            printf("The LENR lock is placed as desired.\n");
            printf("location with address: %p \n\n", fp);
       }
       else printf("ERROR. The LENR lock request was not satisfied.\n");
     }
   }
   return(0);
}          /*  of function lockf                                    */
```

Figure 3 (Part 3 of 3). API Lockf()

This source code is compiled as a module and bound to a service program and a binding directory. After this, this function lockf can be used by any program calling this API and including a different service program or binding directory. The steps involved are:

1. Create the module from the source given in member lockf:

   ```
   CRTCMOD MODULE(MYLIB/LOCKF) SRCFILE(MYLIB/QCSRC) SRCMBR(LOCKF)
   OUTPUT(*PRINT) CHECKOUT(*ALL) DBGVIEW(*ALL)
   ```

2. Create a binding directory:

   ```
   CRTBNDDIR BNDDIR(MYLIB/MYBNDDIR)
   ```

3. Create the binder language source for declaring the items (API lockf in this case) that are to be exported from the service program:

a. Create a source physical file, if required. Add a member MYSRVSRC to it:

```
CRTSRCPF FILE(MYLIB/QSRVSRC) MBR(MYSRVPGM)
```

b. Using SEU, enter the source lines into the member MYSRVSRC:

```
STRPGMEXP  PGMLVL(*CURRENT)
EXPORT   SYMBOL('lockf')  /* export the API name */
ENDPGMEXP
```

The symbol name lockf is enclosed in quotes to protect its case, otherwise symbol LOCKF is exported, which results in an error.

c. Create the service program using the module object lockf (from which symbol is to be exported) and the source binder language code (stored in file QSRVSRC, member MYSRVSRC):

```
CRTSRVPGM SRVPGM(MYLIB/MYSRVPGM) MODULE(MYLIB/LOCKF)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(MYSRVSRC)
```

The service program now has exported symbol lockf.

4. Add information about the service program that is to be associated with the binding directory:

```
ADDBNDDIRE BNDDIR(MYLIB/MYBNDDIR) OBJ((MYLIB/MYSRVPRG))
POSITION(*FIRST)
```

The program in Figure 4 on page 45 is using the lockf API, linking to it using the service program/binding directory previously created. The steps to compile and link it to the lockf API are as follows:

1. Compile the program as a module:

```
CRTCMOD MODULE(MYLIB/MAINLKF) SRCFILE(MYFILE/QCSRC)
OUTPUT(*PRINT) CHECKOUT(*ALL) DBGVIEW(*ALL)
```

2. Link this program to the service program and binding directory that has exported lockf:

```
CRTPGM PGM(MYLIB/RUNLOCKF) MODULE(MYLIB/MAINLKF) ENTMOD(*ONLY)
BNDSRVPGM(MYLIB/MYSRVPGM)
```

or

```
CRTPGM PGM(MYLIB/RUNLOCKF) MODULE(MYLIB/MAINLKF) ENTMOD(*ONLY)
BNDDIR(MYBNDDIR)
```

The runnable program has the name runlockf.

```
    int main(int argc, char *argv·“) {

    int         function, size, result;
    _RFILE      *file_pointer;
    _MOBJL_Template_T  allocated_locks;

    if (argc != 3) {
         printf(″This program needs 2 parameters\n″);
    }

    function = atoi(argv·1“);
    size = atoi(argv·2“);

    if ((function < 0) ||(function > 3)){
         printf(″Value of function is out of range\n″);
         exit(-1);
    }

    if((file_pointer= _Ropen(″MYLIB/MYFILE″,″rr,arrseq=Y″))== NULL){
         printf(″OPEN failed.\n″); exit(-1);
    }

    allocated_locks.Template_Size = sizeof(_MOBJL_Template_T);
    result = lockf(file_pointer, function, size);
    _Rclose(file_pointer);
    printf(″The return value of lockf call is %d \n″, result);
    return(0);
 }
```

Figure 4. Main() Function to Call Lockf()

Chapter 7. Message and Error Handling

In any operating system, communication between procedures or programs, between jobs, between users, and between users and procedures or programs is of great importance. There may be several points within a single application, especially in larger ones (with frequent interactions with the database or between programs) where information regarding the current status of operations has to be passed back and forth. Conventionally, applications are said to pass information to and from, through messages. And therefore, differences in message handling techniques between the source and target systems are of utmost importance in porting applications.

7.1 Message Handling in UNIX

UNIX does not support most of the advanced message handling facilities of the AS/400 system. An application usually interacts with the user or the environment by using built-in C function calls such as *printf(), fprintf(),* and so on, on files and streams that are available on all systems supporting ANSI C, including the AS/400 system. The function *perror()* is used to map the error number in *errno* to an error message for easier debugging.

The system defines three file descriptors inseparably linked with files that it opens and closes automatically. These are:

- stdin (standard input)
  - used by the user to feed data to the application
- stdout (standard output)
  - used by the application to communicate to the user
- stderr (standard error)
  - used by the application to write error messages and warnings to the error file

Alternatively, an application can use displays designed with the help of the **curses** package (defined in the header file *<curses.h>)*, to interact with the user. This again, is easily implemented in ILE C/400 using display files.

Note: On the AS/400 system, file descriptors 0,1, and 2 are not reserved for stdin, stdout, and stderr respectively.

7.2 AS/400 Specific Message Handling Techniques

7.2.1 Message Files and Commands

An application program in the AS/400 system may send messages that are hard coded within the program or stored in the database. An object called a message file (object type *MSGF) stores a list of messages along with their message IDs and severity codes.

The following are the **AS/400 message file commands** for modifying the database of message files:

CHGMSGF
: Change Message File - changes the attributes of a specified message file or a list of message files.

CRTCSPMSGF
: Create CSP/AE User Message File - updates an OS/400 message file with formatted Cross System Product / Application Environment user messages.

CRTMSGF
: Create Message File - creates a user-defined message file for storing message descriptions.

CRTMSGFMNU
: Create Menu from Message Files - creates a menu (display file) from the specified message files.

CRTS36MSGF
: Create S/36 Message File - creates a message file from the System/36 message source member. This enables the user to convert the System/36 message source to the message source on this system.

DLTMSGF
: Delete Message File - deletes the specified message files from the system, including all of the message descriptions stored in the file.

MRGMSGF
: Merge Message File - allows a user to merge messages from one message file with those in another message file.

OVRMSGF
: Override Message File - overrides a message file used in a program. The overriding message file is used whenever a message is sent or retrieved and the overridden message file is specified.

WRKMSGF
: Work with Message Files - allows the user to show a list of message files from one or more libraries.

Control Language (CL) commands may be issued by the application to send messages to the user or to other applications. These commands may have the message text hard coded within them or may refer to a message in a message file with a particular value of the message ID.

The following are the **AS/400 message commands** that may be used to exchange messages with the user or with other applications with or without help from the message file database:

DSPMSG
: Display Messages - used by the display station user to show the messages received at the specified message queue.

MONMSG
: Monitor Message - used to monitor escape, notify, and status messages sent to the program message queue of the program in which the command is used.

RCVMSG
: Receive Message - used by a program to receive a message being sent to a message queue.

RMVMSG
: Remove Message - used by a program to remove the specified message, or a group of messages, from the specified message queue.

RTVMSG
: Retrieve Message - used by a program to retrieve a specified predefined message from a message file and copy it into CL variables in the program.

SNDBRKMSG
: Send Break Message - used to send an immediate message to one or more workstation message queues. The command causes the message to be delivered always in break mode.

SNDMSG
: Send Message - used by a display station user to send an immediate message from the display station to one or more message queues.

SNDNETMSG
: Send Network Message - sends a message to another user on the local or a remote system through the SNADS network.

SNDNWSMSG
Send Network Server Message - sends a message to users or workstations on the Local Area Network (LAN).

SNDPGMMSG
Send Program Message - sends a message to a named message queue or to a call message queue. Each time a program or procedure is called, a new message queue is associated with its call stack entry.

SNDRPY
Send Reply - sends a reply message to the sender of an inquiry message.

SNDUSRMSG
Send User Message - used by a program to send a message to a message queue and optionally receive a reply to that message.

7.2.2 Messages Queues

Message queues provide a form of message passing in which any process (given that it has the necessary permissions) can read a message from or write a message to any message queue on the system.

Note: AS/400 message queues, as discussed here, should not be confused with X/Open IPC message queues. IPC message queues are also supported on the AS/400 system.

The **AS/400 message queue commands** are:

CHGMSGQ
Change Message Queue - changes the attributes of the specified message queue.

CLRMSGQ
Clear Message Queue - clears (removes) all messages from a specified message queue.

CRTMSGQ
Create Message Queue - creates a user-defined message queue and stores it in a specified library.

DLTMSGQ
Delete Message Queue - deletes the specified message queues and any messages in those message queues.

WRKMSGQ
Work with Message Queues - the Work with Message Queues (WRKMSGQ) command shows a list of message queues and

allows the user to display, change, delete, and clear specified message queues.

7.2.3 Message Handling APIs

The message handling APIs let your applications work with AS/400 messages. You can use these APIs to send messages to various destinations, sharing status and error information between programs only, or between programs and users. For details, see the *AS/400 System API Reference*. The **Message Handling APIs** consist of the following:

Change Exception Message (QMHCHGEM)
Changes an exception message on a call message queue. This API allows the current program to perform a variety of actions on an exception message that was sent to its caller, a previous caller, or itself.

Control Job Log Output (QMHCTLJL)
Controls the production of a job log when the related job ends or when the job message queue becomes full and the print-wrap option is in effect for the job.

List Job Log Message (QMHLJOBL)
Lists messages from the job message queue. This function gets the requested message information and returns it in a user space in the format specified in the parameter list.

List Non-program Messages (QMHLSTM)
Lists messages from one or two non-program message queues. This function gets the requested message information and returns it in a user space in the format specified in the parameter list.

Move Program Messages (QMHMOVPM)
Moves messages from one call message queue to the message queue of an earlier call stack entry in the call stack. This is especially useful for error handling.

Promote Message (QMHPRMM)
Promotes an escape or status message that was sent to a call stack entry. That is, the message is handled and replaced with a new escape or status message. You may promote an escape message to another escape message or to a status message. You may promote a status message to an escape message or to another status message.

Receive Non-program Message (QMHRCVM)
Receives a message from a non-program message queue, providing information about the sender of the message as well

as the message itself. This API is similar in function to the Receive Message (RCVMSG) command with the MSGQ parameter.

Receive Program Message (QMHRCVPM)
Receives a message from a call message queue and provides information about the sender of the message as well as the message itself. This API is similar in function to the Receive Message (RCVMSG) command with the PGMQ parameter.

Remove Non-program Messages (QMHRMVM)
Removes messages from non-program message queues. This API is similar in function to the Remove Message (RMVMSG) command with the MSGQ parameter.

Remove Program Messages (QMHRMVPM)
Removes messages from call message queues. This API is similar in function to the Remove Message (RMVMSG) command with the PGMQ parameter.

Resend Escape Message (QMHRSNEM)
Resends an escape message from one call message queue to the message queue of the previous call stack entry in the call stack.

Retrieve Message (QMHRTVM)
Retrieves the message text and other elements of a predefined message stored in a message file on your AS/400 system. This API is similar to the Retrieve Message (RTVMSG) command.

Retrieve Message File Attributes (QMHRMFAT)
Retrieves information about the attributes of a message file.

Retrieve Non-program Message Queue Attributes (QMHRMQAT)
Provides information about the attributes of a non-program message queue.

Retrieve Request Message (QMHRTVRQ)
Retrieves request messages from the current job′s call message queue.

Send Break Message (QMHSNDBM)
Sends a message to a workstation for immediate display, interrupting the workstation user′s task. You can use break messages to warn users of impending system outages and such. This API is similar in function to the Send Break Message (SNDBRKMSG) command.

Send Non-program Message (QMHSNDM)
Sends a message to a system user or a message queue that is not associated with a specific program. This API is similar in function to the Send Program Message (SNDPGMMSG) command with the TOMSGQ parameter.

Send Program Message (QMHSNDPM)
Sends a message to the message queue of a call stack entry in the call stack. This API is similar in function to the Send Program Message (SNDPGMMSG) command with the TOPGMQ parameter.

Send Reply Message (QMHSNDRM)
Sends a response to an inquiry message. This API is similar in function to the Send Reply (SNDRPY) command.

Send Scope Message (QMHSNDSM)
Sends a scope message that allows a user to specify a program to run when your program or job is completed.

7.2.4 Display Files

The most commonly used methodology in the AS/400 system by which an application communicate with the user is through the use of display files. Any use of text windows in UNIX can efficiently be ported to the AS/400 system through display files. A display file defines the format of the information to be presented on a display station, and how that information is processed by the system on its way to and from the display station. Data description specifications (DDS) describe the data referred to by a display file. Messages may be displayed over the entire window, or may take up only a part of the window, thus retaining much of the earlier text. The most popular way of sending messages from an application to a user is through the use of a subfile, which is just a special type of display file.

The following example illustrates the use of a subfile in the AS/400 system for message passing. Figure 5 on page 54 shows a simple DDS for a subfile implementation and Figure 6 on page 54 shows a source file in ILE C/400 that uses this subfile for message passing.

```
A*   The Record format name of the Subfile record is SFL
A*   The Record format name of the Subfile Control record is SFLCTL
A*   The Subfile has two fields NAME and PHONE
A*
A                                           DSPSIZ(24 80 *DS3)
A          R SFL                            SFL
A            NAME             10A    B 10 25
A            PHONE            10A    B   +5
A          R SFLCTL                         SFLCTL(SFL)
A                                           SFLPAG(5)
A                                           SFLSIZ(26)
A                                           SFLDSP
A                                           SFLDSPCTL
A                                    22 25'<PAGE DOWN> FOR NEXT PAGE'
A                                    23 25'<PAGE UP> FOR PREVIOUS PAGE'
```

Figure 5. DDS Source for a Subfile T1520DDG

```
       /*      This program illustrates how to use subfiles.          */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define LEN               10
#define NUM_RECS          20
#define SUBFILENAME       "MYLIB/T1520DDG"
#define PFILENAME         "MYLIB/T1520DDH"


        /*   T1520DDH is a physical file with the following format:   */
        /*             R ENTRY                                        */
        /*               NAME          10A                            */
        /*               PHONE         10A                            */

typedef struct{
    char name[LEN];
    char phone[LEN];
}pf_t;

#define RECLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);
```

Figure 6 (Part 1 of 2). ILE C/400 Source Using Subfile

```
int main(void)
{
    _RFILE            *pf;
    _RFILE            *subf;

/*     Open the subfile and the physical file.                      */

    if ((pf = _Ropen(PFILENAME, "rr")) == NULL)
    {
          printf("can't open file %s\n", PFILENAME);
          exit(1);
    }

    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)
    {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }

/*     Initialize the subfile with records from the physical file.   */
    init_subfile(pf, subf);

/* Write the subfile to the display by writing a record to the subfile control format. */
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadn(subf, "", 0, __DFT);

/*        Close the physical file and the subfile.                  */
    _Rclose(pf);
    _Rclose(subf);
}
void init_subfile(_RFILE *pf, _RFILE *subf)
{
     _RIOFB_T     *fb;
     int          i;
     pf_t         record;

/*      Select the subfile record format.                           */
     _Rformat(subf, "SFL");
     for (i = 1; i <= NUM_RECS; i++)
     {
          fb = _Rreadn(pf, &record, RECLEN, __DFT);
          if (fb->num_bytes != EOF)
          {
               fb = _Rwrited(subf, &record, RECLEN, i);
               if (fb->num_bytes != RECLEN)
               {
                    printf("error occurred during write\n");
                    exit(3);
               }
          }
     }
}
```

Figure 6 (Part 2 of 2). ILE C/400 Source Using Subfile

7.3 Error Handling in AS/400 System

In the AS/400 system, error handling is done through exception handlers, signal handlers, and error handlers.

7.3.1 Exception Handlers

There are three levels of exception handlers in ILE C. They are:

- Direct monitor handlers that are enabled with the #pragma exception_handler directive.
- ILE condition handlers that allow the user to register a condition handler at run time using the ILE condition handler bindable API CEEHDLR.
- HLL-specific handlers, for example, the C signal() function.

The except.h include file declares types and macros used in ILE C/400 exception handling.

# pragma preprocessor directives for Exception handling

- **Exception handler:** Enables a user-defined ILE exception handler at the point in the code where the #pragma exception_handler is located. Any exception handlers enabled by #pragma exception_handler that are not disabled using #pragma disable_handler are implicitly disabled at the end of the function in which they are enabled.
- **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. Any cancel handler that is enabled by a #pragma cancel_handler directive is implicitly disabled when the call to the function containing the directive is finished and the call is removed from the call stack, if the handler has not been explicitly disabled by the #pragma disable_handler directive.
- **Disable handler:** Disables the handler most recently enabled by either the exception_handler or cancel_handler pragma. This directive is only needed when a handler has to be explicitly disabled before the end of a function since all enabled handlers are implicitly disabled at the end of the function in which they are enabled.

For details, see the *ILE C/400 Programmers Guide* and the *ILE C/400 Programmers Reference.*

7.3.2 Signal Handlers

Signals are generated by events, such as system detected errors, programming errors, software interrupts, and user interrupts. The C library routine signal defines how to handle signals. This section lists the different types of signals implemented in both C languages. **Signals Different in the Two Systems**

UNIX supports the following signals that are absent in the AS/400 system. However, this list is not uniform across all UNIX platforms, and the use of a particular signal may differ even on platforms that support them.

SIGEMT	EMT instruction
SIGIOT	IOT instruction
SIGLOST	File lock lost (NFS file locking)
SIGPWR	Power fail
SIGWINCH	Window size change

ILE C/400, in turn, supports a few signals that are not supported in most UNIX platforms. These are enumerated in the following list:

SIGDANGER	System crash imminent.
SIGHUP	Controlling process/terminal is hung up.
SIGINT	System interrupt.
SIGOTHER	Record file error condition.
SIGPOLL	Pollable event.
SIGPRE	Programming exception.
SIGXCPU	Processor time limit exceeded.
SIGXFSZ	File size limit exceeded.

AS/400 Signal Routines Not Present in UNIX

ILE C/400 supports the following signal functions and APIs not present in UNIX C:

QpOsDisableSignals
Disable process for signal

QpOsEnableSignals
Enable process for signal

siglongjump
Perform nonlocal goto with signal handling

sigsetjump
Set jump point for nonlocal goto

For details, see the *AS/400 CPA Process Management Extensions for OS/400 (V3R1)* or the *AS/400 System API Reference (V3R6)*.

7.3.3 Error Handlers

When a C library function or a system call executes unsuccessfully, it may return an error code depending on the reason for failure. This return code is stored in macro **errno** in UNIX C and ILE C/400. Most of the errnos set by UNIX C are available on ILE C/400 through APIs. Most of the unavailable ones are being set by functions for performing operating system related tasks. ILE C/400 does not have such library functions and, hence, these *errno* are not needed.

In the following two tables, we have listed the differences in the *errno* values supported by the two systems. This table points out the errno values in UNIX C that are not supported by or are different in ILE C/400:

Table 10. errno Values in UNIX C That Are Not Supported in ILE C/400

***Errno* of UNIX function**	**Equivalent value in ILE C/400 or AS/400 APIs**	**Description**
ENOEXEC	None	Exec format error
ENOSYM	None	Symbol not in executable
ENOTBLK	None	Block device required
ENOTTY	None	Not a typewriter
ETXTBSY	None	Text file busy
EREFUSED	None	Connection refused
EREMOTE	None	Too many remote in path

The following list gives the errno values in ILE C/400 that are not present in UNIX.

Table 11. errno Values in ILE C/400 That Are Not Available in UNIX C

Errno	**Description**
EBADDATA	Message data is not valid.
EBADKEYLN	The key length specified is not valid.
EBADMODE	Specified file mode is not valid.
EBADPOS	Invalid position.
EBADSEEK	Bad offset.
EGETANDPUT	An illegal write operation after read.
ENOPOS	No record at specified position.
ENOREC	Record not found.
ENOTDLT	File not opened for write.
ENOTOPEN	File not opened for write.
ENOTREAD	File not opened for write.
ENOTUPD	File not opened for write.
ENOTWRITE	File not opened for write.
ENUMMBRS	More than 1 member.
ENUMRECS	Too many records.
EPAD	Padding occurred on write.
EPUTANDGET	An illegal read operation after write.
ERECIO	File is open for record mode.
ESTDERR	Stderr cannot be opened.
ESTDIN	Stdin cannot be opened.
ESTDOUT	Stdout cannot be opened.
ETRUNC	Truncation occurred in I/O operation.

Error Handling Functions in ILE C/400

assert (in assert.h)
Prints diagnostic messages.

atexit (in stdlib.h)
Registers a function to be executed at program termination.

perror (in stdio.h)
Prints an error message to stderr.

_getexcdata (in signal.h)
: Retrieves information about an exception from within a C signal handler.

raise (in signal.h)
: Initiates a signal.

signal (in signal.h)
: Allows handling of an interrupt signal from the operating system.

strerror (in string.h)
: Sets pointer to system error message.

clearerr (in stdio.h)
: Resets the error indicator and end-of-file indicator for the specified stream.

Errors in File Handling

Most errors in file handling are discernible from the return code of the I/O function. For further details about the cause of the error, the I/O Feedback area may be referred to for the File Status code. The _Riofbk function returns a pointer to a copy of the I/O feedback area for the file specified by fp. It returns NULL if an error occurs.

The following example shows how the _Riofbk function may be used.

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
               char name[20];
               char address[25];
               } format1 ;
typedef struct {
               char name[8];
               char password[10];
               } format2 ;
typedef union {
              format1 fmt1;
              format2 fmt2;
              } formats ;
int main(void)
{
_RFILE *fp;                 /*      File pointer                            */
_RIOFB_T *rfb;              /*      Pointer to the file's feedback structure    */
_XXIOFB_T *iofb;            /*      Pointer to the file's feedback area        */
formats buf, in_buf, out_buf;/*      Buffers to hold data                       */
```

Figure 7 (Part 1 of 2). Use of Function_Riofbk()

```
                                 /*Open the device file.                          */
 if (( fp = _Ropen ( ″MYLIB/T1677RD2″, ″ar+″ )) == NULL )
          {
          printf ( ″Could not open file\n″ );
          exit ( 1 );
          }

_Racquire ( fp,″DEVICE1″ );       /* Acquire another device. Replace                */
                                  /* with actual device name.                       */

_Rformat ( fp,″FORMAT1″ );        /* Set the record format for the display file.    */
rfb = _Rwrite ( fp, ″″, 0 );      /* Set up the display.                            */
_Rpgmdev ( fp,″DEVICE2″ );        /* Change the default program device.             */
                                  /* Replace with actual device name.               */
_Rformat ( fp,″FORMAT2″ );        /* Set the record format for the display file.    */
rfb = _Rwrite ( fp, ″″, 0 );      /* Set up the display.                            */
rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf, sizeof(out_buf ));
_Rreadindv ( fp, &buf, sizeof(buf), __DFT );

                                  /* Read from the first device that                */
                                  /* enters data - device becomes                   */
                                  /* default program device.                        */
                                  /* Determine which terminal responded first.      */
iofb = _Riofbk ( fp );
if ( !strncmp ( ″FORMAT1 ″, iofb -> rec_format, 10 ))
if ( !strncmp ( ″FORMAT1 ″, iofb -> rec_format, 10 ))
          {
          _Rrelease ( fp, ″DEVICE1″ );
          }
else
          {
          _Rrelease(fp, ″DEVICE2″ );
          }

                  /*Continue processing*/
printf ( ″Data displayed is %45.45s\n″, &buf );
_Rclose ( fp );
}
```

Figure 7 (Part 2 of 2). Use of Function_Riofbk()

7.4 Preprocessor Directives

Syntax:

#error pp-token“

Error directives on UNIX C produce diagnostic messages only in ILE C/400, it causes the compilation to fail. Therefore, in ILE C/400, you can use the #error directive as a safety check during compilation. For example, if a program uses preprocessor conditional compilation directives, #error directives can be placed in the source file to make the compilation fail if, for instance, a section of the program is reached that should be bypassed.

Chapter 8. Display Handling

This chapter deals with the issues regarding conversion of window components of UNIX applications to the ILE C/400 applications. Porting of user windows are difficult as all I/O operations are generally optimized for the use with a particular architecture. The UNIX C applications are likely to use the curses library, X Windows, and so on for the display handling. In the AS/400 system, the display handling has been made simple with the use of display files, message handling, UIM (User Interface Manager), DSM (Dynamic Screen Manager), and so on. Because of these extra utilities on the AS/400 system, porting display handling components requires changes or rewriting of the code.

8.1 Static and Dynamic Display Handling

Display (or window) handling in UNIX is most commonly done using the curses library routine, though they may not necessarily need the dynamic display management capability. For display interfaces of static attributes, porting is easily done using the display files, subfiles, and UIM. The Screen Design Aid (SDA) is provided in the AS/400 system for generating menus and displays. The SDA allows a user to design displays corresponding to which display files are generated. Using these displays and message files, screen I/O is done the same as any other file I/O in a very simple manner. Refer to Chapter 8 of the *ILE C/400 Programmer's Guide* for further reference of using display files in ILE C/400.

For the applications that have curses-based displays that are defined dynamically, DSM (Dynamic Screen Manager) APIs are to be used in the AS/400 system. These APIs are a set of screen I/O interfaces that provide a dynamic way to create and manage displays for ILE environment high-level languages. The DSM support provided varies from low-level interfaces for direct display manipulation to windowing support. The curses library has functions that have no direct counterpart in DSM. In spite of this, the rewriting or changes are easier than the appearance. The rest of this chapter deals with changing curses application displays to that of DSM. Refer to the *System API Reference* for detailed information about DSM APIs.

8.2 Comparison of Curses and DSM

The difference between curses and DSM is that while curses sets most of the values from the parameters of different available functions, DSM, to some extent, relies on the structures for environment description, window description, and so on to do its job. The comparison is:

Table 12. A Comparison Between Curses and Dynamic Screen Manager (DSM)

Curses	**DSM**
Windows are given various values generally through parameters of a function.	Windows has a unique window description structure, an environment description structure, and so on associated with it. Most of the required attributes are put into these struct from which window picks up various attributes and processing.
Messages are conventionally displayed through a dialogue box that is a new pop-up window.	Every window can have a dedicated message display line apart from the conventional pop-up dialogue box.
Windows are changed using functions that are independent of windows.	Each window can associate certain functions to it that are called when a relevant API call is made. For example, a user-defined draw routine is associated with each window. Whenever an API is called that makes window draw or redraw, this draw routine is automatically called.

In curses, you can take any key as input and process it, such as a command key if required. The method for getting character-at-a-time input without echoing is:
initscr();
nonl(); cbreak(); noecho();
	Character-at-a-time input can be had by defining a character field of length 1. However, this is done through the QsnInpDta() API while command keys are taken through the QsnGetAid() API. These are to be processed separately. Also noecho mode for character-at-a-time input not available.

A default window stdscr is provided. Many functions act on stdscr, while another group of functions takes window handle also as a parameter. For example, addch(ch) acts on stdscr while waddch(win, ch) has window handle also.	There is no default window. All functions use window handle, if required.

8.3 Mapping the Curses Functionality

Mapping of the more commonly used curses functions to that of DSM APIs is given in the following table:

Table 13 (Page 1 of 3). Mapping Curses Functions to Dynamic Screen Manager (DSM) APIs

Curses	**DSM**
*WINDOW *win1;* The preceding statement defines a pointer *win1* that is pointing to a structure of type *WINDOW*.Win1 is thus the window handle.	*Qsn_Win_T win1;* Define a window handle win1.
addch(), addstr(), printw(), mvaddch(), mvwaddch(), mvwaddstr(), mvprintw().	*QsnWrtDta, QsnSetOutAdr()* Almost all output to a window and a display is handled through the QsnWrtDta() API.
All functions that move the cursor to various locations before doing other operations. (For example, functions such as mvaddch()).	The cursor movement functions are not necessarily required as locations are parameter to all *input*, output APIs. However, if required, QsnSetCur() does this job in DSM.
attrof(), attron().	The attributes are passed to the QsnWrtDta() API as a parameter.
beep()	*Qsnbeep()*

Table 13 (Page 2 of 3). Mapping Curses Functions to Dynamic Screen Manager (DSM) APIs

Curses	**DSM**
box(win,vert,hor) That is, box(win1,0,0)	Set the following in window description: border_flag show_border; left_border_char right_border_char right_border_char top_border_char bottom_border_char and so on. For example: Qsn_Win_Desc_T win_desc; /* setting a window des- */ /* cription struct win_desc */ win_desc.show_border = 1; win_desc.border_flag =′1′; win_desc.left_border_char = ′|′; and so on The create window APIs pick up the window attributes from the window description that is passed as a parameter to those APIs. Refer to *System API Reference* for full details.
delay_outputs(ms)	Include this into file: *#include<unistd.h>* Then use *sleep(ms)*.
delch(), mvdelch(), mvwdelch().	Use *QsnWrtDta()*to write blanks at the required positions.
delwin(win), endwin()	*QsnEndWin()*
getchh(), getstr(), mvgetch(), mvgetstr(), mvwgetch(), mvwgetstr()	*QsnGetAID()* gets the command keys, *QsnReadImm(), to read immediate fields QsnReadInp(), QsnReadMDTAlt()* APIs provide the input field read, modified field read capabilities.
getyx() to get the current cursor locations.	*QsnGetCurAdr(), QsnGetCurAdrAID()*
mvwin()	*QsnMovWin(), QsnmovWinUsr()*

Table 13 (Page 3 of 3). Mapping Curses Functions to Dynamic Screen Manager (DSM) APIs

Curses	**DSM**

newwin(). The parameters passed with this function are the position and size of the required window.	*QsnInzWin(), QsnCrtWin(), QsnStrWin().* The position and size of window are picked up from the window description specified in the parameter. For example:
	
	win_desc.top_row = 1;
	win_desc.top_col = 10;
	win_desc.no_row = 5; and so on.

8.4 Example of Changes Required for Porting Window or Display Component

Here is a program that dynamically creates and manages displays using the routines available in curses. The same is changed to run in ILE C/400 applications using DSM APIs that are bindable to ILE C/400 applications. You have to have QSYSINC library, which is separately installable, installed to run DSM with ILE C/400. The changes have been explained by relating the lines of codes that are used for the same processing. The method shown is certainly not the unique method of porting; there are many alternatives.

The program in Figure 8 using the curses library:

```
#include <curses.h>
#include <string.h>
#include <stdlib.h>
#define MAX_STR_LEN 45
#define MAXSCRSIZE 13        /* maximum no. of items in screen    */
                             /* display                           */
int         No_Of_Items
int         Items_In_Scr
char        *heading
char        *Item_name[MAXSCRSIZE];
char        *s="\0"
                             /* Global Window handles             */
WINDOW      *status_win;      6
WINDOW      *dlg_win;

int         rs =0, sel = 0, scrsize =0, done =0;
int         selected_item=0;

int list_files(void){

}

int display_msg( char * msg)   10
{
     char lmsg[MAX_STR_LEN];
     int ret=1, press_len;
     mvwaddstr(status_win,0,2," ");
     mvwaddstr(status_win,1,2," ");
     wrefresh(status_win);
     strcpy(lmsg,msg);
     press_len = strlen(lmsg)+strlen(" Press a key");
     if (press_len <= MAX_SCR_LEN-4)
          strcat(lmsg," Press a key");
```

Figure 8 (Part 1 of 4). Using Curses Functions for Display Handling in UNIX C Environment

```
     else
          mvwaddstr(status_win,1,2," Press a key");
     mvwprintw(status_win,0,2,"%s",lmsg);
     wrefresh(status_win);
     if(wgand so onh(status_win) == 0x1b)
          ret=0;
     mvwaddstr(status_win,0,2," ");
     mvwaddstr(status_win,1,2," ");
     wrefresh(status_win);
     return ret;
}
int list_files(void){

}

int display_menu(int num_row,
                int num_col,
                int pos_y,
                int pos_x ) {

     WINDOW        *w2,*w3;       /* Window handles    6    */
     char          exit_flag = 'n';
     int           cur_item=0 , I=0;
     w2 = newwin(num_row, num_col, pos_y, pos_x);  1
     w3 = newwin(1,78,23,1);
     keypad(w2, TRUE);
     box(w2,0,0);                                 3
     wattron(w2, A_REVERSE);                      4
     rs = strlen(heading);
     mvwaddstr(w2, 0, num_col /2-(rs/2), heading);
     wattroff(w2,A_REVERSE);                      4
     wattron(w3,A_BOLD);
     mvwaddstr(w3,0,25,"ENTER->Select Esc->Exit"); 5
     wattroff(w3,A_BOLD);
     while ((itemname[cur_item] != NULL)
           && (cur_item < MAXSCRSIZE)) {
              strcpy(s, itemname[ cur_item ]);
              mvwaddstr(w2,cur_item+2, 2, s);
              wmove(w2,i+2,2);
              cur_item++;
     }
     wattron(w2,A_REVERSE);
     strcpy(s, itemname[ selected_item ]);
     mvwaddstr(w2,cur_item+2, 2, s);
     wmove(w2,i+2,2);
     wattroff(w2,A_REVERSE);
     wrefresh(w2);
     wrefresh(w3);
```

Figure 8 (Part 2 of 4). Using Curses Functions for Display Handling in UNIX C Environment

```
     while( ! done) {
        switch(wgand so onh(w2)) {
        case 0x1b :/* Esc key was pressed*/    7
           done =1;
           break;
        case KEY_UP :                     7
           selected_item--;
           if(selected_item < 0)
                selected_item = scrsize -1;
           break;
        case 0x0a :/* Selection key ENTER was pressed*/   7
           done=1;
           return(selected_item + 1);
           break;
        case KEY_DOWN :             7
           selected_item++;
           if(selected_item >= scrsize)
                selected_item = 0;
           break;
     }
     werase(w2);
     attrset(0);                        8
     box(w2,0,0);
     wattron(w2,A_REVERSE);
     rs = strlen(heading);
     mvwaddstr(w2,0,num_col/2-(rs/2),heading);
     wattroff(w2,A_REVERSE);

     while ((itemname[cur_item ]  != NULL)
               && (cur_item < MAXSCRSIZE)) {
          strcpy(s, itemname[ cur_item ]);
          mvwaddstr(w2,cur_item+2, 2, s);
          cur_item++;
     }

     wattron(w2,A_REVERSE);
     strcpy(s, itemname[selected_item ]);
     mvwaddstr(w2,cur_item+2, 2, s)
     wattroff(w2,A_REVERSE);                    8
   } /* of While*/
   wrefresh(w2);
   werase(w2);
   wrefresh(w2);
   werase(w3);
   wrefresh(w3);
   delwin(w2);
   delwin(w3);
   return retval;
}
```

Figure 8 (Part 3 of 4). Using Curses Functions for Display Handling in UNIX C Environment

```
int main(void)
{
   initscr();
   status_win=newwin(1,65,21,2);
   dlg_win=newwin(18,70,3,2);
   WINDOW * win1;
   int I =0,choice=0;

   char *msg = "This option is for demo only";
   char *msg3= " This is an invalid option";

   itemname[0] ="1. Delete File";
   itemname[1] ="2. View File";
   itemname[2] ="3. Cancel Spool File";
   itemname[3] ="4. Exit";

   win1 = newwin(23,80,0,0);/* window to put Heading */  9 2

   wattron(win1,A_BOLD); /* this heading is sticked to screen all time */
   box(win1,0,0);
   mvwaddstr(win1,1,26,"SPACE COMMUNICATIONS PVT. LTD.");
   wattroff(win1,A_BOLD);
   wrefresh(win1);                           9

   strcpy(heading,"MAIN MENU");
   while( ! done) {
      sel = display_menu(18,60,3,10)
      if(sel==0) done = 1;
      switch(choice) {
          case 1 :
             list_files();
             break;
          case 2 :
             display_msg();
             break;

          case 3 :
             display_msg();
             break;

          case 4:
             done=1;
             break;

          default :
              display_msg("Invalid option. Internal Error. ");
              break;
      }
   } /* end of while loop */

   werase(win1);
   wrefresh(win1);
   delwin(win1);
   endwin();
}
```

Figure 8 (Part 4 of 4). Using Curses Functions for Display Handling in UNIX C Environment

The following program is the rewritten version in Figure 8 to run in the ILE C/400 environment:

```
/* This program is the using DSM APIs of OS/400 to define and manage       */
/* screens for the applications. The program is functionally equivalent to */
/* the UNIX program given in Figure 8                                      */

#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include ″QCPA/H/pthread″          /* Needed for sleep() function           */
#include ″QSNWIN.H″                /* Header file for DSM APIs              */
#include ″QSNLL.H″                 /*                                       */

             /* Global Declarations                 */
#define MAXSCRSIZE 13              /* maximum no. of items in screen        */
                                   /* display                               */
char *msg, aid ;
int rs=0, sel=0,scrsize=0, done = 0 ;

Qsn_Win_T win1, win2 cur; 1        /* window handles                        */

Qsn_Win_Desc_T win_desc; 1         /* window description for storing */
                                          /* window informations            */
Q_Bin4   win_desc_length = sizeof(win_desc); 1

                                   /* Initializing the function names       */
                                   /* that get associated with various      */
                                   /* window operations                     */
Qsn_Win_Ext_Inf_T ext = { NULL, NULL, NULL, NULL, NULL, NULL }; 1

char *itemname·MAXSCRSIZE“;
char *s = ″\0″;

int selected_item=0;


                                   /* This function is for displaying the   */
                                   /* member list of given file. User can   */
                                   /* delete elect the files for deletion   */
                                   /* from this list                        */

int list_files(void){

}
```

Figure 9 (Part 1 of 5). Using DSM Functions for Display Handling in ILE C/400 Environment

```
                                   /* This generic draw function will       */
                                   /* display a message in the called       */
                                   /* window.  It is called by other draw   */
                                   /* routines.                             */
void GenericDraw(const Qsn_Cmd_Buf_T *cbuf, const Qsn_Win_T *win)
{
   char *msg1 = "F3: quit F4: move  F5: resize F12: cancel";
        5                                       4           4
   QsnWrtDta(msg1, strlen(msg1), 0, -1, 1, QSN_SA_HI, QSN_SA_NORM,
       QSN_SA_BLU, QSN_SA_NORM, *cbuf, *win, NULL);
}           4          4
                                   /* This draw function is used to draw    */
                                   /* a base window                         */
void Draw1(const Qsn_Win_T *win, const Qsn_Cmd_Buf_T *cbuf)
{
  char *heading1 = "SPACE COMMUNICATIONS PVT. LTD.";

  QsnWrtDta(heading1, strlen(txt), 0, 1, 15, QSN_SA_UL, QSN_SA_NORM,
      QSN_SA_RED_RI, QSN_SA_NORM, *cbuf, *win, NULL);
}


                                   /* This draw function is used to draw    */
                                   /* the second window displaying          */
                                   /* the menu.  All the changes in the     */
                                   /* menu (window) is done by              */
                                   /* re-calling this routine.              */
void Draw2(const Qsn_Win_T *win, const Qsn_Cmd_Buf_T *cbuf)
{
  char *heading = "MAIN MENU";
  char *msg2 = "F6: Next Item   F7: Previous Item   F8: Select";
  int cur_item=0;

  GenericDraw(cbuf, win);

 while ((itemname· cur_item “ != NULL)
   && (cur_item < MAXSCRSIZE)) {
   strcpy(s, itemname· cur_item “);
   QsnWrtDta(s, strlen(s), 0, 4+ cur_item, 20, QSN_SA_UL,
       QSN_SA_NORM,QSN_SA_WHT, QSN_SA_NORM,
       *cbuf, *win, NULL);
   cur_item++;
 }
 strcpy(s, itemname· selected_item “ );
 QsnWrtDta(s, strlen(s), 0, 4+ selected_item, 20, QSN_SA_UL,
     QSN_SA_NORM,QSN_SA_WHT_RI, QSN_SA_NORM,
     *cbuf, *win, NULL);
 scrsize = cur_item;                               4          4
 QsnWrtDta( heading , strlen(txt), 0, 1, 20, QSN_SA_UL, QSN_SA_NORM,
             QSN_SA_RED, QSN_SA_NORM, *cbuf, *win, NULL);
                    4          4
    QsnWrtDta(msg2, strlen(msg2), 0, -2, 10, QSN_SA_UL, QSN_SA_NORM,
        QSN_SA_BLU, QSN_SA_NORM, *cbuf, *win, NULL);
}
```

Figure 9 (Part 2 of 5). Using DSM Functions for Display Handling in ILE C/400 Environment

```
int display_menu(int r, int c, int nr, int nc) {

  /* define and start window 2 */

  win_desc.top_row = r; 1
  win_desc.left_col = c; 1
  win_desc.num_rows = nr; 1
  win_desc.num_cols = nc; 1
  win_desc.show_border = ′1′; 3

  win_desc.fullscreen = ′0′;
  ext.draw_fp = Draw2 ;             /* The Function Draw2 is being  */
                                    /* associated with the draw     */
                                    /* routine of win2. QsnCrtWin   */
                                    /* is passed this struct ext    */


  win2 = QsnCrtWin( &win_desc, win_desc_length, &ext, sizeof(ext),
        ′1′, NULL, 0, NULL, NULL);

  cur = win2;

  for (;;) {
    if (( (aid=QsnGetAID(NULL, 0, NULL)) == QSN_F3)) 7
       break;
    else if (aid == QSN_F4)                7
       QsnMovWinUsr(cur, NULL);
    else if (aid == QSN_F5)                7
       QsnRszWinUsr(cur, NULL);
    else if (aid == QSN_F12){              7
       QsnSand so onurWin( win2, NULL);
    }
    else if (aid == QSN_F6) {              7
      selected_item++;
      if ( selected_item == scrsize) {
         selected_item = 0;
      }
      QsnDspWin( win2,NULL); 8
    }
    else if (aid == QSN_F7) {  7
      selected_item--;
      if ( selected_item < 0) {
         selected_item = scrsize -1;
      }

      QsnDspWin( win2,NULL);       8
    }
    else if ( aid == QSN_F8){         7
      return( selected_item + 1);
      break;
    }
    else {
      /* default processing */
    }
  }
}
```

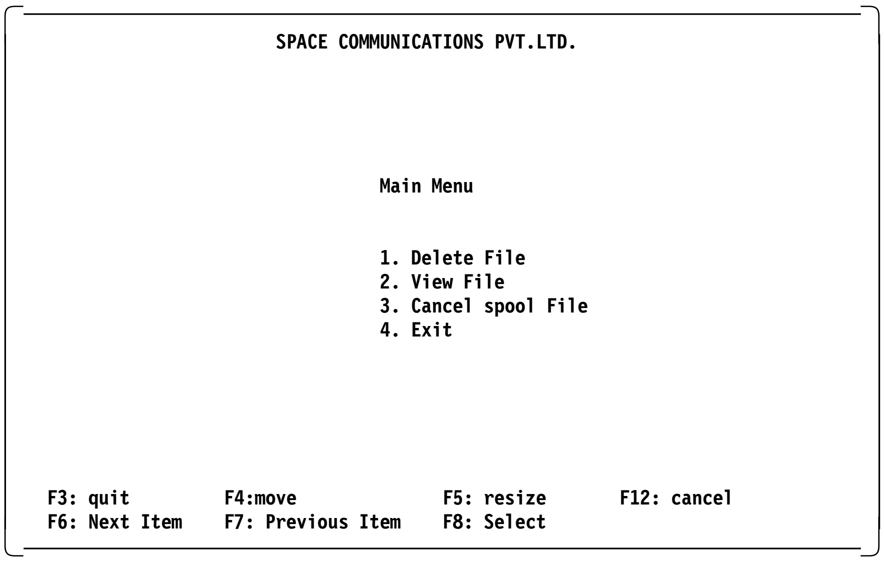
Figure 9 (Part 3 of 5). Using DSM Functions for Display Handling in ILE C/400 Environment

```
int main (void) {
  int i=0;
  char *msg = ″This option is for demo only″;
  char *msg3= ″This is an invalid option″;
  QsnInzWinD(&win_desc, win_desc_length, NULL);

  win_desc.GUI_support = ′1′;

  itemname·0“ = ″1. Delete file″ ;
  itemname·1“ = ″2. View file″ ;
  itemname·2“ = ″3. Cancel spool file″;
  itemname·3“ = ″4. Exit″;

  /* define and start base window */

  win_desc.fullscreen= ′1′;   2

  ext.draw_fp = Draw1;_       9

  win1 = QsnCrtWin(&win_desc, win_desc_length, &ext, sizeof(ext),    9
        ′1′, NULL, 0, NULL, NULL);
  aid = QsnGetAID(NULL,0,NULL);
  while(!done) { /* A non trivial selection was made  */
     sel = display_menu(3,10,18,60);

  if ( sel == 0) done = 1;
     switch ( sel){
      case 1:
        list_files();
        break;

      case 2:
        QsnPutWinMsg( win2, msg, sizeof(*msg),  10
              ′0′, NULL,NULL, 1, 1,
              QSN_SA_WHT,QSN_SA_WHT,
              QSN_SA_NORM,QSN_SA_NORM, NULL);
        done =1;
        sleep(100);
        break;

      case 3:
        QsnPutWinMsg( win2, msg, sizeof(*msg),
              ′0′, NULL,NULL, 1, 1,
               QSN_SA_WHT,QSN_SA_WHT,
               QSN_SA_NORM,QSN_SA_NORM, NULL);
        break;
        done =1;
        sleep(100);

      case 4:
        done=1;
        break;
```

Figure 9 (Part 4 of 5). Using DSM Functions for Display Handling in ILE C/400 Environment

```
      default:
         QsnPutWinMsg( win2, msg3, sizeof(*msg),
              '0', NULL,NULL, 1, 1,
               QSN_SA_WHT,QSN_SA_WHT,
               QSN_SA_NORM,QSN_SA_NORM, NULL);
         sleep(100);
         break;
    }
  }
}
```

Figure 9 (Part 5 of 5). Using DSM Functions for Display Handling in ILE C/400 Environment



```
                         SPACE COMMUNICATIONS PVT.LTD.




                                   Main Menu


                                   1. Delete File
                                   2. View File
                                   3. Cancel spool File
                                   4. Exit






    F3: quit        F4:move             F5: resize       F12: cancel
    F6: Next Item   F7: Previous Item   F8: Select
```

Figure 10. The Output of the DSM Program in Figure 9

Note:

1 Newwin() creates a new window with the specified position in Figure 8. In Figure 9, to initialize a window, three declarations are needed as shown. The position of window is specified through the win_desc struct.

2 A new full screen window is created by making win_desc.fullscreen = '1' in Figure 9 and windows boundaries are not specified in this case.

3 For the box() function in Figure 8, win_desc.show_border = '1' has been specified. In window description, a different character to be used in drawing various window boundaries can be specified, if required.

4 This group of code in Figure 8 is specifying the attribute of output text using attron() and attroff() functions before output function mvaddstr(). In DSM, these attributes are passed with the output function QsnWrtDta().

5 An output is done in Figure 8 using mvaddstr() while in Figure 9, QsnWrtDta() is used.

6 Show window handles in two programs.

7 Processing for input (key_pressed) in the menu. In Figure 9, arrow keys are substituted by F6 and F7. F4, F5, and F12 represent additional facilities shown (just to make the display an AS/400 system type appearance).

8 The group of lines shown in Figure 8 between these marks are for refreshing the window after one iteration of the display loop. It is drawing the window again with the new values attained in previous processing. In Figure 9, this much coding is not required as QsnDspWin() calls the draw function (Draw2 in this case) associated with the window.

9 Shows the codes in two figures that create a base window with a heading.

10 Shows the usual method used for displaying messages. Figure 8 uses a pop-up window. While Figure 9 has put win_desc.msg = '1' (this is the default, hence, not set explicitly in Figure 9). QsnPutWinMsg() is then used to put messages in the window. The pop-up window method can also be used in Figure 9 the same as any other window, if desired.

Appendix A. HP C to ILE C/400 Application Porting

This appendix summarizes the differences that are specific to the HP-UX on an HP 9000 system C language application porting to ILE C/400 on the AS/400 system.

A.1 C Compiler Environment

HP-UX

CC is the HP-UX C compiler, **C89** is the HP-UX POSIX conferment C compiler.

```
cc   {options} files
c89  {options} files
```

The argument ″files″ can be any of the following:

- Arguments ending with .c are C source files, that are compiled and the object file with .o extension is made. If a single C file is compiled and linked all in one step, then the .o file is deleted.
- Arguments with an .s extension, that are assembly source files are assembled and an .o file produced.
- Arguments with an .i extension, that are the output of cpp are compiled without invoking CPP again, and the result put in a file with extension ′.l′.
- Arguments of the form -lx cause the linker to search the library libx. sl or libx.a for resolving currently unresolved external references.
- Other arguments, such as with extensions .o or .a, are considered to be relocatable object files that are to be included in the link operation.

Arguments and options can also be specified to the compiler through the CCOPTS environment variable.

The following options are recognized by both cc and c89:

-n/-N
: Cause the output file from the linker to be marked as shareable/unshareable.

-q/-Q
: Cause the output file from the linker to be marked as demand loadable/not demand loadable.

-s
: Create the output of the linker to be stripped of symbol table information.

-tx,name
: Substitute sub-process x with name, where x is one or more of a set of identifiers indicating the sub-process or sub-processes.

Many compiler options are specific to the HP-UX series; these are:

Series 300/400 Specific Options

+bfpa/ +ffpa/ +M
: Tell the compiler whether to generate code that uses accelerator card / math coprocessor.

+s
: Execute cpp and cpass1 as distinct processes.

-Wc
: Performs function inlining / Causes source code to be printed on the assembly file.

-Wg
: Tell the global optimizer to apply all optimizations.

+y
: Tell the compiler to generate symbolic debugging information or static analysis information for all items.

Series 700/800 Specific Options

+DA / DS
: Generate code and instruction scheduler for the architecture specified.

+FP
: Specifies how the run-time behavior for floating point operations should be initialized at program startup.

+L
: Enable the listing facility and listing pragmas.

+o
: Cause the code offsets to be printed in hexadecimal.

+R num
: Allow only the first num register variables to actually have the register class.

+r
: Inhibits the automatic promotion of float to double.

+u
: Allow pointers to access non-natively aligned data.

+wn
: Specify the level *n* of warning messages.

ILE C/400 The following is a description of the parameters of the CRTCMOD command in ILE C/400 that can be used to approximate the effects of the unsupported HP-UX compiler options discussed previously:

Inline Options
: These allow the user to request that the compiler consider replacing a function call with the called function′s instructions.

Language level
: Specifies which group of library function prototypes are included when the source is compiled.

System includes
: Specifies whether or not the library QSYSINC is added to the library list during compile time. This library provides additional system include files.

Portability Tips for unsupported compiler options:

1. Options -ldir, -Ldir, and -lx should be removed and a corresponding change should be done in *LIBL in OS/400 before compiling.
2. -G option should be replaced by *DEBUG parameter in ILE C.
3. -C option of HP C compiler is similar to *NOGEN of ILE C.
4. -E, -P options are similar to *PPONLY of ILE C.
5. -O is to be replaced by proper option of *PRTFILE parameter of ILE C compiler.
6. -U option should be commented out and #ifdef should be used.
7. Option -n, -N, -q, -Q, -s, and -S should be commented out as these options have functionality not needed in OS/400. If required, equivalent constructs in OS/400 have to be coded to perform the same job.

A.2 Library Functions

1. All of the NLS related functions of HP should be rewritten using the national language support available on ILE C/400. The method to use ILE C NLS is:
   - Create a source physical file with a specific Coded Character Set Identifier (CCSID).

- Change the CCSID of a member in a source physical file to the CCSID of another member in another source physical file.
- Convert the CCSID for specific source in a member.

2. Though all of the mathematical functions of HP C are not available in ILE C/400, almost all functions of HP C can be transferred perhaps with a loss in accuracy. These mathematical functions are mostly variants of the functions that are common to both systems. If they are required in same format, the coding should be done for these functions.
3. **DBM** is a collection of basic database routines available from the C library in HP-C. It can handle only one database at a time. **NDBM** is a collection of upgraded routines and can handle multiple databases. The AS/400 system has an integrated database and its functionality is available to ILE C in the record mode of file operation. These routines are listed under heading **Record Input/Output** of ILE C specific functions in Chapter 3, "File Handling" on page 17.
4. The multimedia audio functions, are not available in OS/400. However, the Ultimedia Facilities feature of OS/400 provides a set of application program interfaces (APIs) to enable AS/400 and programmable workstation (PWS) applications to perform multimedia functions.
5. The *curses* routines of HP C Windowing environment on ILE C can be had by using the bindable APIs. DSM APIs provide the required facility. These functions have to be replaced by DSM APIs bound together with the application.
6. The basic random number generator is available in ILE C. Codes using other variants of functions have to be rewritten using this function.
7. All hardware control functions of **HP** machine are not available in **the AS/400 system.**
8. Packed Decimal set of calls invokes the library functions for emulating 3000-mode (MPE V/E) packed-decimal operations. These functions are in library *libcl.* These functions are not needed in ILE C as packed decimal is a defined data type. The multiplication, addition, and so on can be performed by respective binary operators.
9. Time manipulation in ILE C can be done through CL commands, time manipulation functions, MI functions, and APIs for time manipulation.
10. **Floating point functions** available in HP-C are not used in the AS/400 system, as floating point representations are different in HP and AS/400 machines. The AS/400 system provides **packed decimal** format representation that is listed in Chapter 4, "Networking" on page 21.

11. Library Functions of HP C provide support for managing libraries. The AS/400 system has no equivalent library functions and an object can be used by anyone having proper authority. The related CL commands are in object management and security management.

A.3 Signals

Some of the signals of HP UNIX are not supported on ILE C/400. SIGCLD in HP is for a change in child status, so SIGCHLD should be used in the AS/400 system. Signal for NFS file locking, power fail signal, and window size change signal are other signals that are not supported, but these are not generally required on the AS/400 system.

HP has an additional set of signals that can be used instead of the conventional one. These are*SIGSET, SIGHOLD, SIGRELEASE, SIGIGNORE, and SIGPAUSE.* These are unsupported on other UNIX as well as on AS/400 machines.

A.4 Error Handling

Some of the ERRNO being set in HP C are not supported on ILE C/400. Errors ENOTBLK, ENOTTY, ETXTBUSY, EREFUSED, ENOEXEC, ENOSYM, and EROFS are not supported, hence,they must be monitored by rewriting the code. The ERRNO EROFS (Read only file system error) is replaceable by one from ENOTWRITE, ENOTUPD, or ENOTOPEN in ILE C/400.

A.5 Compiler Directives

Many #pragma can be given in HP C compiler for the manipulation of data alignment. All such #pragma (for example, HP_ALIGN HPUX_WORD, HP_ALIGN POP, and so on) cannot give the data alignment required in ILE C/400. All such #pragmas′ should be commented out during the porting process. Refer to Chapter 2, “Overview of Source and Target Environment” on page 5.

Appendix B. SCO System C to ILE C/400 Application Porting

This appendix summarizes the differences that are specific to the SCO system C language application porting to ILE C/400 on the AS/400 system. The version of the C language supported on the SCO system is commonly known as SCO C or Microsoft (the two terms are used interchangeably in this chapter).

B.1 The C Compiler Environment

The following Compiler Options specific to SCO C are not supported by ILE C/400. Such options, wherever used, should be carefully ported using equivalent commands and options.

-B{1|2|3|a|l|m} *path/filename*
: Defines alternate passes, assembler, loader, preprocessor, and so on.

-compat
: Makes an executable that is binary compatible across the systems--386 UNIX System V Release 3.2, UNIX-286 System V, UNIX-386 System V, UNIX-286 3.0, and UNIX-8086 System V.

-CS{ON|OFF}
: Enables/disables ″common sub-expression″ optimization.

-F *num*
: Sets the size of the program stack to *num* bytes.

-F{a|c|e|l|m|o|s} {*filename***}**
: Changes the default name of the assembly listing, merged assembler and C listing, executable, linker map listing, object, source listing, and so on to filename.

-Gs
: Removes stack probe routines to reduce binary size and speed execution.

-M{s|m|c|l|h}
: Sets the program configuration--that is, defines the memory model, word order, and data threshold.

-n
: Sets pure text model--the only model supported for 80386 binaries.

-nl *len*
: Sets the maximum length of external symbols to *len.*

-nointl
: Directs cc to create a binary that does not include international functionality.

N{D|M|T} *name*
: Sets the data segment name, module name, and text segment name to *name*.

-os2
: Directs cc to create an executable program for OS/2.

-pack
: Packs structures.

-posix
: Enforces strict POSIX conformance.

-S{l|p|s|t} *constant*
: Sets the characters per line, lines per page, subtitle, and title in the source listing.

-strict
: Restricts the language to ANSI specifications.

-unix
: Generates SCO UNIX COFF files.

-W*num*
: Sets the output level for compiler warning messages.

-xenix
: Produces object and executable files using the Intel Object Module Format (OMF).

-xpg3
: Enforces strict XPG3 conformance.

-x2.3
: Produces object and executable files using the Intel Object Module Format (OMF) and the XENIX System V/Release 2.3 run-time library.

-Z{a|d|e|g|i|l|}
: Includes or removes information such as of line numbers, keywords, function declarations, debuggers, libraries, and so on in the output file.

-Zp{1|2|4}
: Packs structure members as specified by #pragma pack() (described later).

Relevant Compiler Options in ILE C/400

The following options during creation of an ILE C/400 module and binding of such modules may be of help where the SCO UNIX compiler options are not supported in the AS/400 system.

Inline Options
These allow the user to request that the compiler consider replacing a function call with the called function′s instructions.

Language level
Specifies which group of library function prototypes are included when the source is compiled.

System includes
Specifies whether or not the library QSYSINC is added to the library list during compile time. This library provides additional system include files.

B.2 #Pragma Compiler Directives

The following pragmas specific to SCO C are not available in ILE C/400. A work-around in the AS/400 system, wherever possible, is mentioned alongside.

- *alloc_text*
  Specifies modules to be grouped into a specified far-text segment.
- *check_stack*
  Controls stack checking on a local basis--that is, by specifying this pragma with parameters *on* or *off*; stack checking for the functions called subsequently may be enabled or disabled.
- *Data_seg*
  Specifies the data-segment name used by functions that load their own data segments. The named segment also contains all data that is normally allocated in the DATA segment.
- *function*
  Specifies which functions are compiled as standard function calls.
- *linesize*
  Sets the number of characters per line in the source listing. On the AS/400 system, the source is listed as a spooled file. The CHGSPLFA can be used to change several attributes of the spooled file.
- *loop_opt*
  Turns loop optimizations on or off. This pragma can be used in conjunction with the [-Ox | -Ol] compiler option with the following effects:

Table 14. Pragma loop_opt() in SCO C

Syntax	**Compiled with the -Ox or Ol Option?**	**Action**
#pragma loop_opt()	No	Turns off optimizations for loops that follow.
#pragma loop_opt()	Yes	Turns on optimizations for loops that follow.
#pragma loop_opt(on)	Yes or no	Turns on optimizations for loops that follow.
#pragma loop_opt(off)	Yes or no	Turns off optimizations for loops that follow.

- *message*
  Sends a message to the standard output without terminating the compilation. On the AS/400 system, the parameters FLAG and MSGLMT are used to approximate the effect.

- *pack*
  This preprocessor directive, along with the -Zp compiler option, controls where and how much packing is done on structure members, as described in the following table:

Table 15 (Page 1 of 2). Pragma pack() in SCO C

Syntax	**Compiled with the -Zp Option?**	**Action**
#pragma pack()	Yes	Reverts to packing specified on the command line for structures that follow

Table 15 (Page 2 of 2). Pragma pack() in SCO C

Syntax	Compiled with the -Zp Option?	Action
#pragma pack()	No	Reverts to default packing for structures that follow
#pragma pack(n)	Yes or no	Packs the following structures to the given byte boundary (n) until changed or disabled

- *same_seg*
  Tells the compiler to assume that specified variables are allocated in the same far data segment.

B.3 C Language Constructs

Portability Tips

Since SCO C closely adheres to the ANSI C standards, the C language is almost identical to that on other ANSI platforms. But it has some specific features, mostly due to hardware dependencies that makes porting of programs from SCO C to ILE C/400 a little different from general porting procedures. The following information is an attempt to highlight these specialties.

- In SCO C, both internal and external identifiers are significant up to exactly 31 characters. In general, C does not specify any length for identifiers and for internal identifiers (including preprocessor directives), at least the first 31 characters are significant; some implementations may make more characters significant. For identifiers with external linkage, implementations may make as few as the first six characters significant.
- The identifiers *asm* and *entry* are deleted from the list of keywords in SCO C, and the keyword *volatile* has been implemented syntactically but not semantically. Instead, some other identifiers can be used as keywords depending on whether the corresponding options are enabled

when the program is compiled. These are: - *cdecl, far, fortran, huge, near*, and *pascal*.

- In SCO C, as a result of the method used to assign types to hexadecimal and octal integer constants, these always act the same as unsigned int in type conversions, whereas UNIX allows octal or hexadecimal integer constants to take types of int, unsigned int, long int, or unsigned long int, depending on its form, value, and suffix.
- SCO C limits the number of hexadecimal digits following ′\x′ in escape sequences to three. It also defines three additional escape sequences, viz.

\v -	represents a vertical tab
\″ -	represents the double-quotation-mark-character
\a -	represents the bell (or alert)

  Also, character constants always have type int, with the result that they are sign extended in type conversions.
- UNIX, in general, does not specify any fixed size for data types int, short int, or long int but only restricts shorts and ints to at least 16 bits and longs to at least 32 bits. Also, shorts cannot be longer than ints, which in turn may not be longer than longs. SCO C defines the short type to be exactly 16 bits long, and long type to be 32 bits long. The size of an int is machine dependent.
- In connection with the *sizeof* operator, the ANSI standards do not enforce any length of a byte, whereas SCO C defines a byte as an 8-bit quantity.
- In general, SCO C type conversions during arithmetic operations are compatible to that in other UNIX versions. However, SCO C describes its conversions in greater detail, including the specific path for each type of conversion.

B.4 Using the Huge Memory Model

Apart from the *near* and *far* type of addressing of ANSI C, SCO C also allows a third type of address--*huge*. The huge address is similar to a far address in that each consists of a segment value and an offset value, but the two differ in the way address arithmetic is performed on pointers. Huge pointers perform pointer arithmetic on all 32 bits of the data item′s address, thus allowing data items to be referenced across more than one segment. The *-Mh* compiler option creates a *huge model program*, wherein both code and data items are accessed with huge addresses, and thus can reside across more than one segment.

The AS/400 system defines all pointers to be 16 bytes long and all SCO C pointers are converted homogeneously to 16 bytes irrespective of the memory model.

B.5 Errno Values

The following SCO specific errno values are not present in ILE C/400:

E2NSYNC, EBADE, EBADR, EBADRQC, EBADSLT, EBFONT, EINIT, EISNAM, ELBIN, ELNRNG, ENANO, ENAVAIL, ENOTNAM, EREMDEV, EREMOTEIO, EUCLEAN, EXFULL.

Hence, sections of SCO C code using these values may have to be modified, possibly by using other similar errnos, or replacing the function calls altogether.

B.6 Signals

There is very little difference between applications on SCO UNIX and those on any other ANSI UNIX platform regarding generation and handling of signals. One notable exception is *sigpoll*, which is issued when a file descriptor corresponding to a stream file has a ″selectable″ event pending. *Sigpoll* is defined to have a value of 20 if used in cross-compiling for XENIX.

Appendix C. Sun Solaris C to ILE C/400 Application Porting

This appendix summarizes the differences that are specific to porting Solaris C applications to ILE C applications. Solaris C is the implementation of C on the Solaris operating system running on an X86 architecture.

C.1 Data Type

Solaris C provides a long data type that is eight bytes in size. It aligns to a boundary of four bytes. This data type is not available in conformance or strictly ANSI mode.

C.2 C Compiler Environment

cc {options} files {libraries}

cc, the Solaris C compiler lets you compile and link any combination of the following:

- C source files having a .c suffix.
- C preprocessed source files, having a .l suffix.
- Operating system object code files, having .o suffixes.
- Assembler source files having .s suffixes.

The following compiler options are recognized by Solaris C in addition to those provided by the standard UNIX compiler:

#
Shows each component as it is invoked (verbose mode).

###
Shows each component as it is invoked but does not execute it.

F
Reserved for floating point operations.

fast
Select the optimum combination of compilation options for speed.

flags
Print a one-line summary of available options.

fnonstd
Causes non-standard initialization of floating point arithmetic software.

fsingle
Causes the compiler to evaluate float expressions as single precision rather than double precision.

fstore
Forces floating point expression assignment to a variable to the precision of the variable.

fnostore
Do not force floating point expressions assigned to the variable to the precision of the variable.

h
Assign a name to a shared dynamic library.

I
Ignore the LD_LIBRARY_PATH setting.

keeptemp
Retain temporary files created during compilation.

native
Generate code for the best floating point option available on the machine.

noqueue
Tells the compiler to not queue your requests if no license is available.

R
A colon separated list of directories used to specify library search directories to the run-time linker.

w
Do not print warnings when compiling.

xF
Enables performance analysis of the executable using Proworks Analyzer and Debugger.

xa
Insert code to count how many times each basic block is executed.

xlibmieee
Force IEEE style return value for math routines in exceptional cases.

xlicinfo
Returns information on the status of licensing.

xM
Run only the macro preprocessor.

xnolib
Do not link any libraries by default.

xnolibmil
Reset-fast option so that it does not include inline templates.

xO
Optimizes for execution time.

xpg
Prepare object code to collect data for profiling with gprof.

xs
Passes -s option to the assembler.

xsb
Generate extra symbol table information for the source code browser.

xsbfast
Create the database for the source code browser but do not compile.

xstrconst
Insert string literals into the text segment.

xtime
Reports the time spent compiling each component.

Portability Tips

1. Options -Idir, -Ldir, and -lx should be removed and a corresponding change should be done in *LIBL in OS/400 before computing.
2. The -g option should be replaced by the DEBUG parameter of CRTCMOD.
3. The -c option of Solaris C compiler is similar to the *NOGEN option of CRTCMOD.
4. The -P options are similar to the *PPONLY option of CRTCMOD.
5. The -O is to be replaced by the OPTIMIZE parameter of CRTCMOD.
6. The DEFINE parameter is the equivalent of -D option.
7. The -xinline option is similar to the *INLINE parameter used in conjunction with # pragma inline.
8. The -w option, specific to Solaris C compiler, can be simulated by setting FLAG=30 for CRTCMOD.

C.3 Signals

The signals SIGWAITING, SIGLWP, SIGFREEZE, SIGTHAW, and SIGCANCEL related to light weight processes and CPR′s are unique to Solaris C and are not supported in ILE C.

C.4 Error Handling

The error numbers unique to Solaris C are ECANCELED, ENOTSUP, EDQOT, EBADE, EBADR, EXFULL, ENOANO, EBADRQC, EBADSLT, EDEADLOCK, and EBFONT. ILE C provides the EDEADLK errno. The rest have to be monitored by rewriting the code.

C.5 Compiler Directives

The following pragmas are Solaris C specific. These can be included within a source file, but may not be used within a function:

#pragma

- **ALIGN**
  Changes the memory alignment of variables to that specified.
- **FINI**
  Calls the functions mentioned after the main function.
- **INIT**
  Calls the functions mentioned before calling main.
- **IDENT**
  Places string in the comment section of the executable.
- **INT_TO_UNSIGNED**
  Changes the return type from unsigned to int.
- **UNKNOWN_CONTROL_FLOW**
  Specifies a list of routines that violate the usual control flow.
- **WEAK**
  Defines a weak global symbol. The linker does not produce an error message if it does not find a definition for the symbols.

These pragmas should be commented out while porting to ILE C as they are not valid pragmas in ILE C. The FINI and INIT pragmas can be simulated by calling the functions at the end and the beginning of main function respectively. A typecasting to int should be done for each call of the functions specified in UNKNOWN_CONTROL_FLOW pragma.

Appendix D. AIX C to ILE C/400 Application Porting

This appendix summarizes the differences that are specific to porting XL C applications to ILE C/400. XL C is the implementation of C on AIX version 3.2 for RISC System/6000. In the rest of this chapter, we use the two names interchangeably.

D.1 Data Alignment

The alignment of data in XL C is as follows:

Table 16. Data Alignment in AIX XL C

Type	Size (bytes)	Alignment of Member Power	Two Byte	Packed
Char	1	Byte	Byte	Byte
Short	2	Half word	Half word	Byte
Long(int)	4	Word	Half word	Byte
Pointer	4	Word	Half word	Byte
Float	4	Word	Half word	Byte
(Long)double	8	Double word if in a union or if first member in a structure, otherwise word	Half word	Byte

The alignment can be changed by using the pragma keyword as well as by compiler options. These are discussed in the following sections.

D.2 C Compiler Environment

The XL C compiler provides the following options in addition to the standard UNIX C compiler. The compiler options can be divided into five categories by function. These are:

Options describing compiler characteristics

-B
Used to construct substitute compiler, assembler, linkage editor, or preprocessor program names.

-qchars=signed/unsigned
Instructs the compiler to treat all variables of type char as having sign type signed or unsigned.

-qcpluscmt/-qnocpluscmt
: Instructs the compiler to recognize the character sequence // as the beginning of a C ++ comment.

-qdbcs
: This option is required if the program contains double-byte characters.

-Fconfig_file:stanza
: This option names an alternate configuration file for xlc.

-qlanglvl=ansi/saal2/saa/extended
: Selects the C language level for the compilation.

-qmbcs
: This option is required if the program contains multibyte characters.

-tprograms
: Designates the programs to which the -B prefix name is appended.

-Wprogram,options
: Passes the listed options to the designated compiler program.

Options describing the compiler object code to be produced

-qalign=power/twobyte/packed
: Sets the alignment of structures and unions.

-qansialias/-qnoansialias
: Specifies whether a type-based alias is to be used during optimization.

-qcompact/-qnocompact
: Reduces code size where possible at the expense of execution speed.

-qenum=int/small
: Controls the storage allocation of enumeration variables.

-qextchk/-qnoextchk
: Generates bind-time type checking information.

-qfloat=options
: Specifies various floating point options.

-qflttrap/-qnoflttrap/-qnoflttrap = options
: Generates extra instructions to detect and trap floating-point exceptions.

-qfold/-qnofold
: Specifies that constant floating-point expressions are to be evaluated at compile time.

-qhsflt/-qnohsflt
: Removes range checking on single-precision float results and on conversions from floating point to integer.

-qhssngl/-qnohssngl
: Specifies that single-precision expressions are rounded only when the results are stored into float memory locations.

-qinitauto=hex-value/-qnoinitauto
: Initializes automatic storage to the hexadecimal byte value hex_ value.

-qisolated_calls=names
: Lists functions that do not alter data objects visible at the time of the function call.

-qmaf/-qnomaf
: Specifies whether floating-point multiply-add instructions are to be generated.

-qmaxmem=size
: Limits the amount of memory used for local tables of specific, memory-intensive optimizations to size kilobytes.

-O3/-O2/-O/-qoptimize=3/-qoptimize=2/-qoptimize/-qnooptimize
: Optimizes code at a choice of levels during compilation.

-qproclocal [**= names**]**/-qprocimported** [**= names**]**/-qprocunknown** [**= names**]
: These options mark functions as local, imported, or unknown.

-Q / -Q! / -Q-names / -Q+names / -Q=threshold
: Attempts to inline functions instead of generating calls to a function.

-qrndsngl/-qnorndsngl
: Specifies that the result of each single-precision (float) operation is to be rounded to single precision.

-qro/-qnoro
: Specifies the storage type for string literals as read-only/read-write storage.

-qrrm/-qnorrm
: Prevents floating point optimizations that are incompatible with run-time rounding to plus and infinity modes.

-qspill=size
: Specifies the register allocation spill area as size entries.

-qspnans|-qnospnans
: Generates extra instructions to detect signalling NaN on conversion from single precision to double precision.

-yrounding_mode
: Specifies the compile time rounding mode of constant floating point expressions.

Options describing the compiler output

-qattr/-qattr=full
Produces a compiler listing that includes an attribute listing. The default is noattr.

-qflag=severity1:severity2
Specifies the minimum severity level at which diagnostic messages are reported.

-qhalt=severity
Stops compilation at any compilation phase that encounters an error of a specified severity or greater.

-qlist/-qnolist
Produces a compiler listing that includes an object listing.

-qlistopt/-qnolistopt
Produces a compiler listing that displays all options in effect.

-qnoprint
Suppresses listings. It overrides all listing options.

-qsource/-qnosource
Produces a compiler listing and includes C source code.

-qsrcmsg/-qnosrcmsg
Specifies the style of diagnostic messages.

-qstat/-qnostat
Produces a compiler listing that reports table size and timing statistics.

-w
Requests that warning messages be suppressed. This option is equivalent to flag=e:e.

-qxref/-qxref=full/-qnoxref
Produces a compiler listing and includes a cross-reference listing.

Options used for debugging

-#
: Traces compilation.

-qcheck/-qnocheck
: Causes the program to generate trap information for run-time exceptions that the dbx symbolic debug program uses to determine the cause of the exception.

-qdbxextra/-qnodbxextra
: Specifies whether the -g option is to generate information in the object file for all symbols or only for those that are referenced.

-qignprag=disjoint/isolated/all
: Ignores either or both of the disjoint and isolated_call alias pragmas.

-pg
: Sets up the object files for profiling, but provides more information than is provided by the -p option.

-qphsinfo/-qnophsinfo
: The phsinfo reports the time taken for both the entire compilation and each compilation phase.

Options performing preprocessor functions

-M
: Creates an output file that contains targets suitable for inclusion in a description file for the AIX make command.

-ma
: Substitutes inline code for calls to function alloca.

Options used by the linkage editor

-r
: Permits the output file to be produced even though it contains unresolved symbols

Portability Tips

1. Setting the OPTION parameter of CRTCMOD to *XREF/*NOXREF is equivalent to -qxref/-qnoxref options.
2. A -qnoprint can be achieved by setting the OUTPUT parameter to none.
3. OPTIMIZE = BASIC/FULL corresponds to -O3/-O2 options.
4. The equivalence in inlining compiler options in the two systems is:

   -Q! is the same as INLINE = OFF.

-Q = threshold is equivalent to INLINE = ON, MODE = AUTO and THRESHOLD = threshold.
-Q + names can be achieved by specifying #pragma inline for the named functions in the source program.
-Q - names can be achieved by specifying #pragma no inline in the source program.

5. A -qlanglvl is equivalent to the LANGLVL parameter of the CRTCMOD command.
6. The lower limit of -qflag can be achieved by setting the FLAG parameter.
7. A -qhalt is equivalent to the MSGLMT parameter.

D.3 Pre-Processor Directives

Some pragmas available in XL C are:

```
#pragma options compiler_options
#pragma  langlvl
#pragma  chars
#pragma  strings
#pragma  isolated_call
#pragma  disjoint
```

The chars and langlvl pragmas are available in ILE C as well.

D.4 CPI-C on AIX

CPI-C is the programming Interface to APPC on the AIX system. AIX SNA Services/6000 running on all models of the IBM RISC System/6000 workstation supports application program use of CPI communications calls to communicate with programs on other RISC System/6000 workstations or on other systems in an SNA network. Communication with other programs on the same RISC System/6000 workstation is supported in a limited fashion.

The CPI communications calls are part of the library libcpic.a that is shipped with SNA Services/6000. These calls are structured to act as independent C-language calls. SNA Services/6000 offers some extensions to the basic CPI communications interface. These extension calls have a prefix of xc rather than the cm prefix used for the basic CPI Communications calls.

Deviations from CPI C on OS/400

CPI communications calls on SNA Services/6000 are supported by the C language only.

SNA Services/6000 supports CPI communications calls with these distinctions:

- The Set_Log_Data call is accepted; however, it performs no operation. SNA Services/6000 does not log or transmit the data.
- The Set_Return_Control call accepts only the value CM_WHEN_SESSION_ALLOCATED. This means that control does not return to the program until a conversation is allocated.
- Communication with other programs on the same RISC System/6000 workstation LU is not supported.
- Communication with other programs on the same RISC System/6000 workstation, but on different LUs, is supported if the workstation is configured so that each LU is associated with a different control point on the workstation.

Appendix E. DEC ALPHA C to ILE C/400 Application Porting

This appendix summarizes the differences that are specific to the DEC ALPHA C language application porting to ILE C/400 on the AS/400 system.

E.1 Data Types and Alignments

Data types in OSF/1 differ from other UNIX as well as from the AS/400 system in the size of pointer type and long. The data types that differ are:

Table 17. Data Alignment in DEC Alpha C

Data Types	**Size in bytes**
long	8
long long	8
pointer	8

Data alignment is implied by data type. For example, an int (32 bits) is aligned on a 4-byte boundary; a long (64 bits) is aligned on an 8-byte boundary.

The DEC ALPHA C compiler supports the use of 4-bytes pointers on the 64-bit DEC OSF/1 operating system. All system interfaces use 64-bit pointers. The 4-byte pointer data type is provided to help developers reduce the amount of memory used by dynamically allocated pointers, and to assist with the porting of applications that contain assumptions about the sizes of pointers.

The use of 32-bit pointers in applications requires source code modifications and the use of compiler options.

- Short pointer: A 32-bit pointer.
- Long pointer: A 64-bit pointer. This is the default pointer type on DEC ALPHA systems.
- Simple pointer: A pointer to a non-pointer data type, for example: int *num_val;.
- Compound pointer: A pointer to a pointer, or a pointer to an indefinite array, for example: char *argv• or char **FontList.

Two cc flags and a set of pragmas control the usage of 32-bit pointers. The compiler flag - also causes the compiler to respond to the #pragma pointer_size directives. The -xtaso_short compiler flag causes the compiler to allocate 32-bytes pointers by default and is recognized only when used with the -xtaso flag. The cc flags for controlling pointer size are the following:

- -xtaso: Enables the use of short pointers. All pointer types default to long pointers, but short pointers can be declared through the use of the pointer_size pragmas.
- -xtaso_short: Enables the use of short pointers. All pointer types default to short pointers. Long pointers can be declared though the use of the pointer_size pragmas.

E.2 C Compiler Environment

On DEC ALPHA, machine C has two versions of compilation mode:

- The default is OSF/1 C.
- DEC C is obtained by the ***-migrate*** flag.

cc -option(s) filenames

The various compilation flags are grouped into:

General options

-cpp
Runs the C macro preprocessor on C and assembly source files before compiling.

-Dname

-Dname=def
Define a name to the C macro preprocessor by a #define directive. If no definition is given, the name is defined as 1.

-I
Causes the C macro preprocessor to never search for #include files in the standard directory (/usr/include).

-Idir
Causes the C macro preprocessor to search for #include files whose names do not begin with a slash (/) in dir after looking in the current dir but before looking in the standard directory.

-Ldir
Specifies the pathname dir as an additional search directory for the linker.

-no_cpp
Does not run the C macro preprocessor on C and assembly source files before compiling.

-P
Runs only the C macro preprocessor and puts the result for each source file in a .i file.

-non_shared
Produce a static executable program.

-o outfile
Specifies that the executable program is named outfile rather than the default, a.out.

-o output
Names the final output file output.

-std
Enforces the ANSI standard with extensions.

-std0
Enforces the K&R standard with some ANSI extensions.

-std1
Enforces the ANSI standard.

-v
Displays the compiler passes as they execute.

-verbose
Displays the long form of error and warning messages.

-w or -w1
Suppresses warning messages.

-w2
Displays warnings and aborts as if an error occurred.

-w3
Suppresses warning messages. Exits with a non-zero status when warnings occur.

Options for macros

-resumption_safe
All -scope_safe functionality plus the code generated in trap shadows are restricted so that they are re-executable.

-scope_safe
Ensures that any trap pc is reported in the procedure or guarded scope the trap occurred in.

-shared
Produces a shared object.

-call_shared
: Produces a dynamic executable program file. The object created may use shared objects at run time.

-lstring
: Specifies additional libraries to search in addition to the libraries associated with the compiler driver invoked on the command line. The characters specified as string are appended to lib and form a file name of a library.

-migrate
: Enables language processing rules and language extensions for the DEC C compilation environment.

Debugging options

-g or -g2
: Permits full source-level debugging. These flags often suppress optimizations that might interfere with full debugging.

-g0
: Produces an object file without debugging information, thereby reducing its size; use when debugging is no longer required. It also retains all optimizations. This is the default.

-g1
: Permits accurate but limited source- level debugging, retains most optimizations.

-g3
: Permits full but inaccurate debugging on fully optimized code. The debugger output may be confusing or misleading.

Profiling options

-no_pg
: Disables gprof profiling for all objects that follow this flag on the cc.

-p or -p1
: These flags set up profiling by periodically sampling the value of the program counter and affect only the linking.

-p0
: Disables profiling. This is the default.

-pg
: Sets up gprof profiling that sets up statistical sampling, call graph reporting, and links with grt0.o and libprof1.a.

Optimizer options

-O0
Prevents all optimizations.

-O1
Causes the assembler and the code generator to perform as many optimizations as possible without affecting compile-time performance.

-O3, -O4
Performs global register allocation across the bounds of individual compilation units.

-Olimit num
Specifies the maximum size in basic blocks of a routine that is optimized by the global optimizer.

-om
Invokes the post-link optimizer.

-O or -O2
Global optimization. Optimizes within the bounds of individual compilation units.

Options to control floating point operations

-fprm c
Enables chopped rounding (round towards zero) of the results of IEEE floating-point instructions generated by the compiler.

-fprm d
Enables the dynamic setting of the rounding mode applied to the results of IEEE floating-point instructions.

-fprm m
Enables the rounding to minus infinity of the results of IEEE floating-point instructions generated by the compiler.

-fprm n
Enables normal rounding (unbiased round to nearest) of the results of IEEE floating-point instructions generated by the compiler. This is the compiler′s default behavior.

-fptm n
Disables all floating point trapping modes.

-fptm su
Enables floating point trapping on underflow with software completion.

-fptm sui
Enables floating point trapping on underflow or inexact with software completion.

-fptm u
Enables floating point trapping on underflow.

-ieee_with_inexact
Provides full IEEE support.

Default flags that are in effect during compiler invocation: std0, cpp, call_shared, g0, O1, p0 **Portability Tips:**

1. Option -g0 is to be replaced by DBGVIEW(*NONE) in ILE C/400.
2. -P, -cpp is the same as *PPONLY of ILE C/400.
3. A -o is to be replaced by the PGM parameter of ILE C.
4. All -Dname are to be replaced by #ifdef in ILE C/400.
5. -w, -w1, -w2, and -w3 are related to the warning generation. Replace by appropriate value of FLAG parameter in ILE C/400.
6. ILE C/400 follows ANSI standard. Hence, all compilations correspond to -std1. -Std0 (non ANSI compilation) is not allowed though -std is supported but for the extensions.

E.3 Macros

The following macros are defined in ALPHA C. The table shows the compilation modes in which these are supported:

Table 18. Macros used with DEC Alpha C Compiler

Macro	**std0**	**std**	**std1**
LANGUAGE_C	Yes	No	No
__LANGUAGE_C__	Yes	Yes	Yes
UNIX	Yes	No	No
__UNIX__	Yes	Yes	Yes
__osf__	Yes	Yes	Yes
__alpha__	Yes	Yes	Yes
LANGUAGE_ASSEMBLY	Yes	Yes	Yes
__LANGUAGE_ASSEMBLY_	Yes	Yes	Yes

E.4 Preprocessor Directives

Using Pragmas:

A pragma is a preprocessor directive, similar in syntax to a #define statement. The #pragma directive has the following form: #pragma token-sequence

The C compiler supports the following pragmas:

Weak
The weak pragma defines a new weak external symbol and associates this new symbol with an external symbol.

Pack
The pack pragma is used to change the alignment restrictions on structure members.

Intrinsic
Intrinsic functions are functions in which the C compiler generates optimized.

Function
Code in certain situations, possibly avoiding a function call to to control whether or not a function is treated as an intrinsic you use one of hte following pragmas (the func_name_list is a comma-separated list of function names that optionally is enclosed within parentheses):
#pragma intrinsic [(] func_name_list [)]

#pragma function [(] func_name_list [)]

#pragma function ()

Pointer_size
This pragma has the following syntax:

#pragma pointer_size specifier
The specifier must be one of following:

- Long
- Short
- Save
- Restore

Preprocessor directives enabled by the -migrate flag: Compiling the -migrate flag with the cc driver enables additional preprocessor directives that are available only with this flag. The following pragmas are supported:

- [**no**]**inline**
  Function inlining is the inline expansion of function calls; it replaces the function call with the function code itself.

- [**no**]**member_alignment**
  By default, the compiler aligns structure members on natural boundaries.

  #pragma [no]member_alignment {specifier} preprocessor directive to determine the byte-alignment of structure members. This pragma has the following formats:

```
#pragma member_alignment
#pragma member_alignment save
#pragma member_alignment restore
#pragma nomember_alignment
```

- **Message**

  The #pragma message directive controls the issuance of individual diagnostic messages or groups of diagnostic messages. Use of this pragma overrides any command-line flags that may affect the issuance of messages.

  The #pragma message directive has the following format:

  ```
  #pragma message flag1 (message-list)
  #pragma message flag2
  ```

E.5 Error Handling

Following are errno values that are different in DEC ALPHA C:

EBADMSG, EBADRPC, EBUSY, ECANCELED, EDIRTY, EDOM, EDQUOT, EINVAL, EIO, EMTIMERS, ENODATA, ENODEV, ENOENT, ENOEXEC, ENOSR, ENOSTR, ENOSYM, ENOTBLK, ENOTTY, EPERM, EPROCUNAVAIL, EPROGMISMATCH, EPROGUNAVAIL, EPROTO, ERANGE, EREMOTE, ERPCMISMATCH, ETIME, ETOOMANYREFS, EUSERS, EVERSION

The DEC OSF/1 Calling Standard for AXP Systems defines special structures and mechanisms for the processing of exceptional events. The standards defined are the following:

- The manner in which exception handlers are established.
- The way in which exceptions are raised.
- How the exception system searches for and invokes a handler.
- How a handler returns to the exception system.
- The manner in which the exception system traverses the stack and maintains procedure context.

A termination handler consists of code that executes when the flow of control leaves a specific body of code.

The ability to raise user-defined exceptions or convert UNIX signals to exceptions are through routines such as exc_raise_status_exception, exc_raise_signal_exception, exc_raise_exception, exc_exception_dispatcher, and exc_dispatch_exception. These exception management routines also provide the mechanism to dispatch exceptions to the appropriate handlers. In the case of C-language, the structured exception handling capabilities provided by the DEC ALPHA C compiler allow you to deal with the possibility that a certain exception condition may occur in a certain code sequence. The syntax establishing a structured exception handler is as follows:

```
try {
       try-body
       }
except (exception-filter) {
        exception-handler
        }
```

This has to be substituted by appropriate codes in ILE C/400. Refer to the *ILE C/400 Programmers Guide* for details of exception handling in ILE C/400.

E.6 Signals

The signal SIGTKSZ set by ALPHA is not available in ILE C/400. The signal SIGPWR can be mapped to SIGDANGER of ILE C/400 in most situations.

Appendix F. 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 systems. The information in this publication is not intended as the specification of any programming interfaces that are provided by the OS/400, 5716-SS1, and Common Programming APIs. See the PUBLICATIONS section of the IBM Programming Announcement for 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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

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

IBM	IBM
Application System/400	AS/400
C/400	FORTRAN/400
Operating System/400	OS/400
OS/2	Sustem Application Architecture
SAA	SQL/400
400	

Other trademarks are trademarks of their respective companies.

F.1 Industry Standards

The ILE C/400 library routines are designed according to the specifications of the American National Standard Programming Language C, American National Standard Institute (ANSI), and the IBM System Applications Architecture (SAA) Common Programming Interface for the C language.

Appendix G. Related Publications

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

G.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How To Get ITSO Redbooks" on page 123.

- *UNIX C Application Porting to AS/400*, SG24-4438-00
- *The IBM AS/400 as a TCP/IP Network File Server*, GG24-4092-00
- *IBM AS/400 V3 Communication API Handbook*, SG24-2573-00
- *Developing (Real) DCE Applications for OS/400*, SG24-2572-00
- *AS/400 Integrated Language Environment: A Practical Approach* , GG24-4148-00
- *IBM AS/400 TCP/IP Configuration and Operation*, GG24-3442-02

G.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	**Subscription Number**	**Collection Kit Number**
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RISC System/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RISC System/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
Application Development Redbooks Collection	SBOF-7290	SK2T-8037
Personal Systems Redbooks Collection (available soon)	SBOF-7250	SK2T-8042

G.3 Other Publications

These publications are also relevant as further information sources:

- Version 3 Release 6 publications:
  - *AS/400 CL Programming*, SC41-4721
  - *AS/400 CL Reference*, SC41-4722

  - *AS/400 Common Programming APIs Toolkit/400 Reference*, SC41-4802
  - *AS/400 ILE C/400 Programmer's Guide*, SC09-2069
  - *AS/400 ILE C/400 Programmer's Reference*, SC09-2070
  - *AS/400 ILE Concepts*, SC41-4606
  - *AS/400 Integrated File System Introduction*, SC41-4711
  - *AS/400 Machine Interface Functional Reference*, SC41-3810
  - *OS/400 Data Management*, SC41-4710
  - *OS/400 Work Management*, SC41-4306
  - *AS/400 Sockets Programming*, SC41-4422
  - *AS/400 System API Reference*, SC41-4801
- Version 3 Release 1 publications:
  - *AS/400 CL Programming*, SC41-3721
  - *AS/400 CL Reference*, SC41-3722
  - *CPA Extensions for OS/400 Reference*, SC41-3820
  - *CPA Process Management Extensions for OS/400*, GI10-1000
  - *AS/400 ILE C/400 Programmer's Guide*, SC09-1820
  - *AS/400 ILE C/400 Programmer's Reference*, SC09-1821
  - *AS/400 ILE Concepts*, SC41-3606
  - *AS/400 Integrated File System Introduction*, SC41-3711
  - *OS/400 Data Management*, SC41-3710
  - *OS/400 Work Management*, SC41-3306
  - *AS/400 Sockets Programming*, SC41-3422
  - *AS/400 System API Reference*, SC41-3801
- Others:
  - *APPC Programming*, SC41-3443
  - *Programming Reference Summary*, SX41-3720
  - *SAA CPI - Communications Reference*, SC26-4399-06
  - *System API Programming*, SC41-3800
  - *TCP/IP configuration and Reference*,SC41-3420
  - *TCP/IP File Server Support/400 Installation and User's Guide*, SC41-0125

G.4 References on the World Wide Web (WWW)

Internet users may find information at the following World Wide Web sites:

AS/400 home page: http://as400.rochester.ibm.com

ITSO Redbooks home page: http://www.redbooks.ibm.com

AS/400 Partners in Development home page:
http://www.softmall.ibm.com/as400
In particular, click on ″Of Interest to UNIX Developers″

How To Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at URL http://www.redbooks.ibm.com.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** — to order hardcopies in United States
- **GOPHER link to the Internet** - type GOPHER.WTSCPOK.ITSO.IBM.COM
- **Tools disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get lists of redbooks:

  ```
  TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
  ```

  To register for information on workshops, residencies, and redbooks:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  ```

  For a list of product area specialists in the ITSO:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
  ```

- **Redbooks Home Page on the World Wide Web**

  http://w3.itso.ibm.com/redbooks

- **IBM Direct Publications Catalog on the World Wide Web**

  http://www.elink.ibmlink.ibm.com/pbl/pbl

  IBM employees may obtain LIST3820s of redbooks from this page.

- **ITSO4USA category on INEWS**
- **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**

  With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** (Do not send credit card information over the Internet) — send orders to:

	IBMMAIL	**Internet**
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** — send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** — send orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1) 415 855 43 29 (Outside USA)** — ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **Direct Services** - send note to `softwareshop@vnet.ibm.com`

- **On the World Wide Web**

Redbooks Home Page	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Internet Listserver**

  With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to `announce@webster.ibmlink.ibm.com` with the keyword `subscribe` in the body of the note (leave the subject line blank).

IBM Redbook Order Form

Please send me the following:

Title	**Order Number**	**Quantity**

- **Please put me on the mailing list for updated versions of the IBM Redbook Catalog.**

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

- Invoice to customer number

- Credit card number

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

DO NOT SEND CREDIT CARD INFORMATION OVER THE INTERNET.

Glossary

Advanced Program-to-Program Communication (APPC). The APPC support handles all of the SNA protocol requirements when your system is communicating with a remote system using the LU type 6.2 and node type 2.1 architectures. You can connect your system to any other system that supports the APPC program interface.

American Standard Code For Information Interchange (ASCII). It is the coded character set used by UNIX. It defines 128 characters and each is represented by 7-bit binary values.

API. See *Application Program Interface.*

APPC. See *Advanced Program to Program Communication*.

Application Program Interface (API). A formal interface that is intended to be used in the building of applications. It provides a path into system functions.

argument. In a high-level language (HLL) procedure call, an expression that represents a value that the calling procedure passes to called procedure.

ASCII. See *American Standard Code For Information interchange*. ASCII is the coded character set. It defines 128 characters and each is represented by 7-bit binary values.

bind. To create a program that can be run by combining one or more modules created by an Integrated Language Environment (ILE) compiler. See also *binder* and *binding.*

binder language. A small set of commands (STRPGMEXP, EXPORT, and ENDPGMEXP) that defines the external interface (signature) for a service program. These commands are in a source file and cannot be run alone.

binder. The system component that creates a bound program by packing Integrated Language Environment (ILE) modules and resolving symbols passed between those modules.

binding directory. A list of names of modules and service programs that may be needed when creating an ILE program or service program. A binding directory is not a repository of modules and service programs. Instead, it allows them to be referred to by name and type.

binding. The process of creating a program by packaging an Integrated Language Environment (ILE) modules and resolving symbols passed between those modules.

block. A group of one or more logical records treated as a single piece of data.

blocked record. A physical record that contains more than one logical records.

bound program. An AS/400 object that combines one or more modules created by an Integrated Language Environment (ILE) compiler.

call message queue. A message queue that exists for each call stack entry within a job.

call stack entry. A program or procedure in the call stack. Each call stack entry has information about the local, automatic variables for the procedures, and other resources scoped to the call stack entry such as condition handlers and cancel handlers.

call stack. The ordered list of all programs or procedures currently started for a job. The order is last in, first out. The programs and procedures can be started explicitly with the CALL instruction, or implicitly from some other event.

call. Adds a new entry to the call stack for the called procedure or program and transfers control to the called object.

CCSID. See *Coded Character Set Identifier.*

character set. The aggregate of all valid characters allowed by the standard followed.

child process. A new process created by an existing process. The new process is thereafter known to the preexisting process as the child process. The preexisting process is called parent process.

client (Xwindow). An application program connects to the window system server by some Interprocess communication (IPC) path, such as a TCP connection or a shared memory buffer. This program is referred to as a client of the window system server.

client. A node in a network that requests a service to be performed.

Coded Character Set Identifier (CCSID). A set of values used by the AS/400 system to denote different languages.

collation. The logical ordering of strings in a predetermined sequence according to established rules. This is usually done to provide native language support.

command language interface. The interface between the HLL and the OS that processes all commands issued by the HLL program and generates equivalent code for the OS and vice-versa.

commit. To make all changes permanent that were made to one or more database files since the last commit or rollback operation, and to make the changed records available to other users.

commitment control. A means of grouping file operations that allow the processing of a group of database changes as a single unit through the Commit command or the removal of a group of database changes as a single unit through the Rollback command.

commitment definition. Information used by the system to maintain the commitment control environment throughout a routine step and, in case of a system failure, throughout an initial program load (IPL). This information is obtained from the Start Commitment Control command, which establishes the commitment control environment, and the file open information in a routing step.

connection (Xwindows). The IPC path between the server and client program is known as a connection.

connectivity. Interaction between the client and server.

control language. The system language on the AS/400 that provides greater control of lower level system features than HLLs.

conversion descriptor. A structure used by AS/400 Data Conversion APIs to pass encoded information about the conversion method being followed.

CPI-C. CPI Communications provides a consistent application programming interface for applications that require program-to-program communication. The interface makes use of SNAs LU 6.2 to create a rich set of inter-program services.

curses. A C subroutine library to allow programs to easily display output at specific positions on the screen of a cathode ray tube (CRT). It uses the termcap terminal capabilities database. Curses and termcap were originally developed for the *vi* text editor.

daemon. Daemon processes on the UNIX system do system-wide functions, such as administration and control of networks. They are not associated with any user.

data alignment. Data of a particular size needs to be stored on the storage boundaries with addresses that are a particular multiple of byte addresses. The resultant arrangement of data is called data alignment.

data description specification (DDS). A description of the user′s database or device file that is entered into the system in a fixed form. The description is then used to create files.

data type modifier. Keywords that affect the allocation or access of data storage. The two data type modifiers are *const* and *volatile.*

database file. An AS/400 file type used to store data.

DBCS. See *Double-Byte Character Sequence.*

debug mode. A mode in which a program provides detailed output about its activities to aid a user in detecting and correcting errors in the program itself or in the configuration of the program or system.

debug. To detect, diagnose and eliminate errors in programs.

debugger. A tool used to detect and trace errors in computer programs.

default entry point. An entry point in an application that receives control from the operating system. In a C program, the *main()* function is the default entry point.

descriptor. A set of data structures that collectively represent the characteristic of an open file. The data structures contain the file′s attributes, identification, access control, accounting information and are maintained by file system routines.

direct monitor handler. An exception handler that allows the application programmer to directly declare an exception monitor.

display file. An AS/400 file type for screen I/O.

Double-Byte Character Sequence (DBCS). A sequence of two bytes used in the AS/400 system to denote characters of native languages that are not supported by the EBCDIC character set.

dynamic screen manager. A set of APIs for controlling screen interaction.

EBCDIC. See *Extended Binary Coded Decimal Integer Code.*

enumeration. A data type in C language that can have a value from a specified set of values.

environment variables. Strings of the form ″name=value″ that are stored in an environment space outside the program.

error handler. An operating system facility to trap errors and pass relevant information to the application identifying the error.

event (Xwindow). Clients are informed of information asynchronously by means of events.

exception handler. An operating system facility to trap exceptions and pass relevant information to the application identifying the exception.

exception handling. Handling of error condition by the application.

export. An external symbol defined in a module or program that is available for use by other modules or programs. See also *external symbol.* Contrast with *import.*

Extended Binary Coded Decimal Integer Code. (EBCDIC). It is the coded character set used by the AS/400 system. It defines 256 characters and each is represented by 8-bit binary values.

external data. Data that is exported from one procedure and imported by another procedure. Contrast with *internal data.*

external symbol. An item defined in a high-level language program that represents such things as procedures or variables. Resolving external symbols is the means by which the binder connects modules to form a bound program or a service program.

externally described data. Data contained in a file for which the fields and the records are described outside the program using facilities such as DDS, IDDU, and SQL/400, and used by the program when the file is processed.

file descriptor. In the UNIX environment, the integer that identifies a file.

IFS. See *integrated file system.*

ILE. See *Integrated Language Environment.*

import. A reference to an external symbol defined in another module or program. Contrast with *export.*

indicator. An internal switch used by a program to remember when a certain event occurs and what to do when that event occurs.

integrated file system (IFS). The file system in OS/400 that supports stream I/O and storage management similar to UNIX while providing an integrating structure over all of the information stored on the AS/400 system.

Integrated Language Environment (ILE). A set of constructs and interfaces (APIs) for all ILE conforming HLLs.

Interprocess communication (IPC). Communication between different processes.

I/O feedback area. An area in the memory allocated to a job where the AS/400 system stores information about all I/O operations done in the current job.

IPC. See *interprocess communication.*

keyed sequence access path. An access path to a database file that is arranged according to the contents of key fields contained in individual records. See also *arrival sequence access path* and *access path.*

linger timer. The time to wait before any buffered data to be sent is discarded during close() socket operation.

locale. An object that contains information about valid alphabetic characters, the collating sequence, the format of numbers and currency amounts, and the format of date and time. This information is used by certain C/400 library routines in application programs.

locking. The technique of capturing access to a shared resource, wherein any other process or job can be completely or partially restricted from using it.

logical file. A description of how data is to be presented to a program. This type of database file contains no data, but it defines the format for one or more physical files.

machine interface. The interface between the OS and the hardware. Limited control is available to the programmer on this interface through system functions and APIs.

memory model. A basis for classifying memory based on its size. This also determines the size of the memory addresses.

menu. An AS/400 object for screen I/O where the user can optionally choose to enter a system command as well.

message. A means of communication between the job and the user. Types of message are inquiry, status, escape, informational, notify and so on.

message file. An AS/400 object for storing message texts along with unique message IDs.

message description. All information about a particular message - message ID, message text, severity, type, and so on.

message queue. A list on which messages are placed when they are sent to a person or program. The AS/400 system-recognized identifier for the project is *MSGQ.

module. In the ILE model, module is the object that results from compiling the source code. Module cannot be run before binding into a program object or service program.

mutex. A mutual exclusion mechanism for synchronizing between threads, such as a fast memory lock.

native language support. A feature of OS/400 that provides tools for customizing applications for International use.

network. A computer network is a collection of computer nodes physically connected by a suitable communications medium.

network protocols. Network protocols are sets of rules that control the communication and transfer of data between two or more devices in a communications system.

NFS. A file sharing protocol with the goal of providing file sharing in a heterogeneous environment.

OOB data. Out-of-band (OOB) data is user-specific data that only has meaning for connection-oriented (stream) sockets.

open data path (ODP). A control block created when a file is opened. An ODP contains information about merged file attribute and information returned by input or output operations. The ODP only exists while the file is open.

packed decimal. A method of storing decimal data in a more compact form using half-byte for each digit and another half byte for the sign.

padding. Unused spaces in the data file to align the different type of member of structure or union at their alignment boundaries. This alignment is required to provide faster retrieval of data.

parameter. (1) In ILE, an identifier that defines the type of arguments that are passed to a called procedure. (2) A value supplied to a command or program that is used either as input or to control the actions of the command or program.

physical file. A description of how data is to be presented to or received a program and how data is actually stored in a database. A physical file contains one record format and one or more members.

pointer. A pointer type variable holds the address of a data object or a function.

pragma. An implementation-defined instruction to the compiler.

preprocessor. A section of the compiler that enables you to modify the default compilation process by changing the inputs to the compiler or modifying the steps in the compilation process.

procedure call. A call made to a procedure within a module in a bound program. See also *static procedure call* and *procedure pointer call.* Contrast with *program call.*

procedure pointer call. A high-level language call mechanism for specifying the address of a procedure to be called. The procedure pointer call provides a way to call a procedure dynamically. For example, by manipulating arrays or tables of procedure names or addresses, the application programmer can dynamically route a procedure call to different procedures. Contrast with *static procedure call.*

procedure. A set of self contained high-level statements that perform a particular task and then return control to the caller.

program call. A call made to an ILE program or to an OPM program. See also *dynamic program call.* Contrast with *procedure call.*

program. In the ILE model, the runnable objects that result from binding modules together.

promote. To convert an unhandled condition into a new condition with a different meaning. The new condition is passed on to another condition handler.

record I/O. The file I/O method where all data stored in files is treated as a number of records, which may contain one or more fields. Contrast with *stream I/O.*

referential integrity. Homogeneity between the parent file and its dependent files.

resource (Xwindow). Windows, pixmaps, cursors, fonts, graphics, contexts, and colormaps are known as resources. They all have unique identifiers associated with them for naming purposes.

resource. Any device or item used by a computer, for example, I/O devices, disk files, or programs.

return. To remove the call stack entry and transfer control back to the calling procedure or program in the previous call stack entry.

SAA. See *System Application Architecture (SAA).*

SBCS. See *Single-Byte Character Sequence.*

semaphore. A synchronization primitive, similar to a mutex or a machine interface (MI) lock. It can be used to control access to shared resources, or used to notify other processes of the availability of resources.

server (Xwindow). The server provides the basic windowing mechanism. It handles IPC connections from clients, demultiplexes graphics requests onto the screens, and multiplexes input back to the appropriate clients.

server. The node in a network that handles the clients request.

service function. Functions provided by the system for debugging or adjusting the performance of CPA programs.

service program. A bound program that performs utility functions that can be called by other bound programs. See also *bound program.*

shared memory. That part of the virtual address space, which is shared by more than one process, so that they can directly communicate amongst themselves by reading or writing into this space.

shell. The program (in UNIX) that controls user interactions with the system and executes commands.

signal. A way by which the operating system handles asynchronous events.

Single-Byte Character Sequence (SBCS). The default form of data encoding on the AS/400 system, where a single byte is used to store one character. See also *Double-Byte Character Sequence.*

SNA. See *System Network Architecture.*

sockets. Sockets allow you to write your own applications to supplement those supplied by the network protocol. Sockets allow unrelated processes to exchange data locally and over networks.

source entry utility (SEU). A function of the AS/400 Application Development Tools licensed program that is used to create and change source members.

spooled file. Spooling allows the system to store data in an object called a spooled file. The spooled file collects data from a device until a program or device is available to process the data. A program uses a spooled file as if it were reading from or writing to an actual device. Input Spooling is done by the system for database and diskette files. Output Spooling is done for printers.

stored procedure. A remote procedural call that does not require the compilation of interface definitions, or creation of stub programs.

stream I/O. The file I/O method where all data stored in files is treated as a sequence of characters. Contrast with *record I/O.*

subfile. A type of display file on the AS/400 system with special features.

System Application Architecture (SAA). An architecture defining a set of rules for designing a common user interface, programming interface, applications programs, and communications support for strategic systems such as OS/2, OS/400, and MVS/370 operating systems.

Systems Network Architecture (SNA). This is the dominant protocol in IBM based networks.

TCP/IP. Transmission Control Protocol/Internet Protocol refers to a family of non-proprietary network protocols, of which TCP, providing host-to-host transmission, and IP, providing data routing from source to destination, are two important parts.

thread. A technique for concurrent programming by allowing multiple flows of processing within a process. Each thread in a process is a separate processing flow.

transaction. A group of individual changes to objects on the system that should appear as a single atomic change to the user.

translator. An OS/400 component that performs the final step in a program or module compilation. In the ILE model, the translator is called the optimizing translator.

trigger. A trigger is a set of actions that are run automatically when a specified change operation is performed on a specified physical database file. The change operation can be an insert, update, or delete high-level language statement in an application program.

trigraphs. Sequences of three characters used to emulate characters used in the C language that are not available on some IBM keyboards.

UEP. See *user entry procedure.*

UIM. See *user interface manager.*

user interface manager. An AS/400 system utility to dynamically manage displays and user interfaces.

window handle. A pointer to a structure of type WINDOW, that is automatically defined by the system for each window created. It contains information about all attributes of the window.

X-windows. A facility to create, modify, and manage graphical user interfaces, independent of, but supported by UNIX.

Index

Printed in U.S.A.

SG24-4938-00