IBM

International Technical Support Centers

AS/400 C LANGUAGE INTRODUCTION

GG24-3434-00

AS/400
C Language
Introduction

Document Number GG24-3434

NOV 1989

International Technical Support Center
Rochester Minnesota

First Edition (Nov 1989)

This edition applies to Release 2.0 of the 5728-CX1 C - Language Compiler for use with the OS/400.

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 program product in this document is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

The information contained in this document has not been submitted to any formal IBM test and is distributed on an 'As Is' basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Publications are not stocked at the address given below. Requests for IBM publications should be made to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Center
Dept. 977, Building 003
Rochester, MN 55901 USA

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Abstract

This document describes some functions of the AS/400[1]
IBM C Language Compiler 5728-CX1. It provides information on the product announced in late 1989 which will have controlled availability in early 1990. The information for this book has been obtained by working with the pre-release product in the Toronto Lab. In addition much of the information was obtained by talking with the planning and development group for this product. Because of the timing of the research for this book it may or may not be a true indication of the final product when it is released at general availability time. Some information will become obsolete as later releases of the product become available.

This document is intended to provide introductory planning information for persons considering the IBM C Compiler on the AS/400. It is intended to help in the evaluation of the use of the AS/400 C Compiler and run time environment. There are many cases where a comparison is made to a similar function on the AS/400 done with another language such as RPG III or COBOL. There are many coding examples to show how some functions are done in C. If the reader is knowledgeable with another language then he can compare the coding effort in C with the other language.

A detailed knowledge of C is NOT necessary to understand this document. However introductory knowledge of both C and the AS/400 will help with some of the terminology and concepts.

All of the coding examples have been compiled and run on an AS/400. However since the testing was performed on a pre-release system there is no guarantee that the final compiler available to the user will perform exactly as our test system did.

C/400 (128 pages)

[1] Application System/400 and AS/400 are trademarks of International Business Machines Corporation.

Acknowledgments

The advisor for this project was:

Lamont Baker
International Technical Support Center, Rochester

This publication is the result of a residency conducted at the Toronto Languages Lab in June-July of 1989. Thanks are due to the three residents who worked on this project :

Klaus Subtil - IBM Germany
David Choi - IBM Hong Kong
Sandria Rosin - IBM Canada

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

- The Toronto Languages C Development group for time and patience in explaining many of the concepts that are new to an AS/400 person. Thanks are also due to this group for providing a working environment which allowed the residents to write and test programs in C/400 in addition to facilities to document this book.
- The writers who are working on the C/400 reference manuals for providing early copies of reference guides.
- The Application Testing group for assistance in some of the earlier coding exercises.
- The Performance Group for ideas in areas that need further investigation and explanation.
- The Business Partners group for early feedback and expected acceptance of the C/400 product in specific application areas.

Preface

This document is intended for persons requiring an introductory understanding of the C language as used on the AS/400. This document explains the various levels of C available on the AS/400 and the implications of using basic ANSII C, SAA[2] Level 2 or other AS/400 extensions to C.

The purpose of this document is to show how functions are performed with AS/400 C. It explains why the AS/400 needs a special environment to run C language programs. It explains how C language functions are combined to create an application environment. This document gives examples of standard ANSII C functions or AS/400 extensions. For example, there are coding examples of data base access using standard C function calls and further examples using the AS/400 SQL extensions. Similarly there are examples using standard C screen sessions and further extensions that use AS/400 DDS described display files.

Through the use of examples the reader is able to compare the coding required and output available with AS/400 C to the coding required and output available with other AS/400 languages.

If the reader is familiar with C on other platforms, then this document should serve to show how some of the SAA and AS/400 extensions are used.

The document is organized as follows:

- "Introduction"

  This chapter explains some of the strengths of the C language and the various levels of use for C on the AS/400.

- "Application Development Considerations"

  This chapter gives some application development considerations when using C on the AS/400. It explains the character set differences that C will require and some of the different environment considerations when using C.

- "AS/400 C Environment"

  This chapter expands on the special environment required for the use of C. It explains why this environment is needed and how it differs from that needed for other programming languages.

- "Data Manipulation"

  This chapter describes the differences between the data handling with C and other high-level languages on the AS/400. It shows how to have a C program interface to the standard DDS screen files and database files on the AS/400 as well as the use of SQL from C.

- "Display File Processing"

  This chapter shows how the AS/400 program interfaces with a display file and explains the requirement for a generic data stream handler program.

---

[2] Systems Application Architecture and SAA are trademarks of International Business Machines Corporation

- "Performance Conclusions and Considerations"

  This chapter gives the results of preliminary testing comparing the C language to some other AS/400 languages. It shows the difference in performance to be expected in database operations and in simple numeric situations. Also included are the source programs that were used to reach these conclusions. Included in this chapter is a section which summarizes the results from a construct analysis group in the language development laboratory.

- "Sample C Programs"

  This chapter has sample programs to show the use of AS/400 C for simple functions or to explain the use of other standard AS/400 extensions.

Related Publications

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

Order Number	Title
SC09-1303	Languages: C/400 User's Guide
GC09-1304	IBM AS/400 Licensed Program Specifications: C/400
SC09-1308	SAA CPI C Reference - Level 2
SC09-1316	Application Performance Tuning Aid: User's Guide and Reference
SC21-9608	Structured Query Language/400 Reference
SC21-9609	AS/400 SQL/400 Programmer's Guide
SC21-9620	Programming: DDS Reference
SC21-9659	Programming: Data Base Guide
SC21-8079	Programming: Backup and Recovery Guide
GG24-3321	SQL/400 A Guide to Implementation
GG24-3354	ITSC Redbook: SAA Portability Guidelines
SC26-4348	SAA CPI Database Reference
SX09-1139	Languages: C/400 Reference Summary
GC09-1312	C/400 Runtime Library PRPQ Licensed Program Specifications

Contents

1. Introduction 1
1.1 What is C? 1
1.2 Strength of the C Language 2
1.3 Levels of C Language on AS/400 and Portability 3

2. Application Development Considerations 5
2.1 Character Set Required 5
2.2 How to Set Up PC 5250 Emulation for C Characters 5
2.3 Trigraphs 6
2.4 Source Conversion from Other Systems 6
2.5 Distinct Upper and Lower Case Names 7
2.6 Character vs Record Processing 7
2.7 Secured Pointer Usage 7
2.8 User-Defined Data Stream 7
2.9 Floating Point Implementation 8
2.10 Return Codes in C/400 Programs 8

3. AS/400 C Environment 11
3.1 Environment Introduction 11
External Variables 12
Multiple Entry Points 12
3.2 Extended Program Model 13
3.3 EPM Environment 13
3.4 Debug 16
3.5 Exception Handler 17
3.6 Session Manager 19
3.7 EPM Application Library 20
3.8 SETPGMINF 22
3.9 EXTPGMINF 22
3.10 Application Performance Tuning Aid 23
3.11 Location of C/400 Runtime Routines 24

4. Data Manipulation 25
4.1 Alignment of Data 25
A Method to Determine Record Length 25
Referencing Numeric Elements in Structures 26
Processing an AS/400 Externally Described File with C/400 27
Use of a Logical File to Map Fields 27
4.2 Stream Mode Data 30
4.3 Text Stream Mapping on AS/400 to Source File 30
Example of Writing Characters to a Text File 31
Specifying Record Length in a Text File 33
Text File and Binary File are the Same from AS/400 Viewpoint 33
Major Differences in Processing Text File and Binary File 33
4.4 Accessing an Externally Described File in a C/400 Program 33
Character String 33
Numeric Data 35
C/400 Program and Record Format of Externally Described File 35
C/400 Files or AS/400 Externally Described File? 36
4.5 How to Use AS/400 Database Files in C 36
Using Existing AS/400 Data 41

Usage of Logical Files for Data Type Overrides 42
4.6 Open Query File Usage 43
4.7 Commitment Control 44
4.8 Coding SQL - Considerations of Usage 46
C Host Variables for SQL 46
4.9 Using Existing Data in SQL/400 Tables 47
Static SQL in a C/400 Program 49
Using Dynamic SQL in C/400 50

5. Display File Processing 57
5.1 Usage of Indicators 57
Using Indicators with the INDARA Keyword in Display Files 58
Using Indicators as Data in a Display File Input/Output Buffer 59
5.2 Use of Conversion Routines 60
5.3 Print Key 61
5.4 Display File Handling in a C/400 Program 61
Display File and QXXFORMAT 61
Display File Without Using Indicator 61
Display File Using Separate Indicator Area (INDARA) 61
Display File with Indicator as Data in Input/out Buffer 61

6. Performance Conclusions and Considerations 63
6.1 General Observations from Performance Comparison Programs 63
6.2 Performance Observations from Coding Analysis Tests 64
6.3 Test programs 66
List of Test Programs 66
6.4 Main Program 67
Display File of Main Program - TSCREEN 67
Database File Used by Main Program - FSCREEN 67
COBOL Main Program 67
6.5 TEST01 - C/400 Program to Write Record to File with fwrite 69
6.6 TEST02 - COBOL Program to Write Record to File 70
DDS for Database File FSAM11A 72
6.7 TEST03 - C/400 Program to Write Record to File with QXXFORMAT . . . 72
DDS for Database File FSAM11B 73
6.8 TEST04 - C/400 Program to Write Record to File with fwrite and Move Fields 73
6.9 TEST05 - COBOL Program to Write Record to File and Move Fields . . 75
6.10 TEST06 - C/400 Program on Arithmetic Operation with Binary (Integer) 76
6.11 TEST07 - COBOL Program on Arithmetic Operation with Zone Decimal 77
6.12 TEST08 - C/400 Program on Arithmetic Operation with Floating Point . 78
6.13 TEST09 - COBOL Program on Arithmetic Operation with Binary 79
6.14 TEST10 - COBOL Program on Arithmetic Operation with Pack Decimal 80
6.15 TEST11 - C/400 Program call COBOL for Random Record Read 81
COBOL Program Called to Read File 83
6.16 TEST12 - C/400 Program with SQL for Random Record Read 84
6.17 TEST13 - COBOL Program with Random Record Read 86
Database File Used in Program BIGF1L 88
6.18 TEST14 - COBOL Program with SQL for Random Record Read 88
TEST15 - C/400 Program with strncpy 90
6.19 TEST16 - C/400 Program with memcpy 91

Appendix A. Sample C Programs 93
A.1 Sample Programs 93
A.2 Example 1 - PGM01 (String Substitution) 93

A.3 Example 2 - PGM02 (Unsigned Packed Field to Signed Packed Field Conversion) 97
A.4 Example 3 - PGM03 (Writing Records to Program-Described File with C) 101
A.5 Example 4 - PGM04 (Reading Records from a File by C) 104
A.6 Example 5 - PGM05 (Read Record from AS/400 Database File) 105
A.7 Example 6 - PGM06 (Working with Database and Display File) 107
A.8 Example 7 - PGM07 (Working with Printer File) 110
A.9 Example 8 - PGM08 (Working with Display File with INDARA) 113
A.10 Example 9 - PGM09 (Writing Records to a Database File with C) . . 116
A.11 Example 10 - PGM10 (Dynamic SQL under Commitment Control) . . 120

Index 129

1. Introduction

1.1 What is C?

The C language was developed in 1972 by Dennis Ritchie of Bell Labs to function with the UNIX operating system. It was based on the earlier B language written by Ken Thompson, also from Bell Labs. In the early 1980s commercial implementations of C for personal computers became available. As a result C became the most popular PC implementation language.

In 1983 the American National Standards Institute (ANSI) committee (X3J11) for C was formed to standardize the language. This ANSI C Standard was finalized in 1989. Coding according to the ANSI Standard will allow your C code to be more easily portable across competitive vendor platforms. Aside from the ANSI C standard, there is also a Systems Application Architecture (SAA) definition for the C programming language. This SAA definition is based on the draft ANSI Programming Language C (CX3J11/88-090), dated December 7, 1988.

The SAA definition includes a set of software interfaces, conventions and protocols, and provides a framework for designing and developing applications with cross-system consistency. SAA provides a framework across the following IBM computing environments:

- TSO/E in the Enterprise Systems Architecture/370
- CMS in the VM/System Product or VM/Extended Architecture
- Operating System/400 (OS/400)[3]
- Operating System/2 (OS/2)[4]
  Extended Edition
- IMS/VS Data Communications in the Enterprise Systems Architecture/370
- CICS/MVS in the Enterprise Systems Architecture/370

Further information on the SAA standards can be found in *System Application Architecture, Common Programming Interface, C Reference - Level 2.*

The C/400 programming language implements SAA Level 2 language on the AS/400 system. It is ANSI C with SAA extensions. Options in the AS/400 C compiler allow you to identify non-ANSI or non-SAA functions in a program. This should facilitate the generation of code which is portable over multiple environments. In addition, there are AS/400 extensions to C which would be appropriate in environments where portability to other platforms is not a consideration.

---

[3] Operating System/400 and OS/400 are trademarks of International Business Machines Corporation

[4] Operating System/2 and OS/2are trademarks of International Business Machines Corporation

1.2 Strength of the C Language

C is a low-level language which offers great flexibility. One of its greatest strengths has been its portability. This is achieved in part by the way C applications are generally written. A C program is usually composed of many short C routines or functions combined together. Functions that depend on hardware or operating system implementations are normally coded separately and stored in a library to be called by the application programs. This isolates the machine-dependent code from the application code. Standard routines for I/O, string manipulation, calculations, etc. are normally included with a C compiler and have standard function names. So the application merely calls the appropriate function from the standard system library when needed.

Although C offers low-level control such as bit manipulation, as you would expect from an assembler language, it also supports modern program structures. These include statements such as If (then), While, For, Do, as well as data structures and arrays. The data types supported are character, integer, floating point (real) and pointers. C/400 will allow programmers to take advantage of C's high-level control and data structures while still affording some of the capabilities of assembler language programming.

Although C provides great flexibility and control, it should be noted that this environment has few run time checks compared to the traditional RPG or COBOL environment. For instance, the programmer is responsible for verifying that data types are appropriate for a specified operation, and that end-of-string characters are inserted and checked for in string processing. There are no level checks when accessing database files, so the programmer must insure that the structure defined within a C program maps correctly to the current version of the database file. Precision in these areas is required or programmer productivity and maintainability of code could be adversely affected. In short, the C language is a powerful tool but is geared more for the professional developer rather than the casual programmer.

The C/400 library includes many functions for input and output, mathematics, exception handling, string and character manipulation, dynamic memory management, and date and time manipulation. The library helps to maintain program portability, since the machine-dependent details for the various operations need not concern the programmer.

In addition, C programs can call high-level language (HLL) programs such as RPG, COBOL and PL/I; this is not usually supported on other systems. C programs can be called from HLL programs as well. This allows you to code a routine in the language most appropriate and call it from your application program. Routines that might have been written in assembler language on other machines, can be written in C on the AS/400.

C/400 should be of significant interest to customers who wish to convert existing C language applications to the AS/400 system. Also, as mentioned earlier, C can be used for routines that would have been written in assembler language on other machines. Finally, the C/400 programming language should be considered for accounts that have a requirement for portable application development or a requirement to support IBM SAA environments and AS/400-architected solutions.

1.3 Levels of C Language on AS/400 and Portability

The C/400 programming language is SAA Level 2; this is ANSI C with SAA extensions. An example of an SAA extension is SQL support. Normal ANSI C would use stream I/O and not be concerned with SQL. If you are developing an application which will run on IBM and non-IBM platforms, you would want to restrict your code to ANSI C in order to make it more easily portable. If, however, you will be developing code for various IBM platforms, you could use the SAA extensions and not limit your portability. Options in the AS/400 C compiler allow you to identify non-ANSI or non-SAA functions in a program.

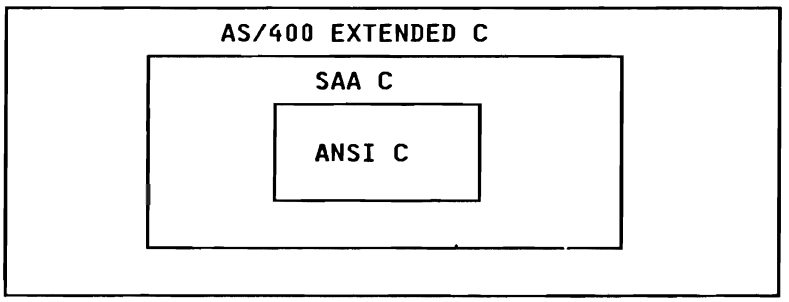


Figure 1-1: Levels of C on AS/400

The most current definition of SAA is level-2 and this is the level of implementation of C on the AS/400. The SAA definitions are still evolving, and some of the functions that we would like to see available in C/400 are not yet defined in SAA (for example, keyed I/O and relative record number (RRN) data base access). However, these functions are planned for a future release of C/400.

In addition to the ANSI and SAA functions, there are AS/400 extensions to C that can be used to take advantage of some of the capabilities unique to the AS/400. For example an AS/400 C application may use the commit/rollback functions of the AS/400. Using these extensions would enhance your application on the AS/400, but naturally, inhibit the portability of the code. There are also native AS/400 functions that are not available in the C/400 extensions such as support for zoned decimal and packed data without conversion to floating point or integer and using externally defined data and display files without redefining the layout within the C program. However, packed structures and externally defined display and data files are planned for a future release. There is more information on accessing database files and using display files in subsequent chapters of this document.

2. Application Development Considerations

2.1 Character Set Required

The C language uses an extensive character set. In program development mode the user may require access to characters which are either not on his keyboard or are not displayable on his terminal.

These will include the 52 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 10 digits:
0 1 2 3 4 5 6 7 8 9
and the following graphics:
! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] _ { } ~
In addition, the bitwise exclusive OR in ASCII as ^ or EBCDIC as ¬
and the ¦ or |
on some systems it may be desirable to have the Horizontal Tab, Vertical Tab, Form Feed, and End of String as keyable characters.

2.2 How to Set Up PC 5250 Emulation for C Characters

Neither the older 5250 Emulation Program nor its older 5250 Emulation Card may be used for C since neither the keyboard nor the display can be modified with this program.

If the IBM Personal Computer Enhanced Display Station Emulation Adapter Card is used with the Enhanced 5250 Emulation Program then the keyboard profile will have to be modified to include statements which allow keying the hexadecimal values for the square brackets.

A real 5250 keyboard does not have square brackets and when using 5250 emulation a PC will have to emulate the use of the hex key followed by the hex values for open and closed brackets. The following example shows the two statements which must be added to the profile which is referred to by the emulation program. For example they may be located in the KBPC.PRO file. In the sample profile statements shown below the alternate key followed by the square brackets on a PC keyboard are treated as the hex key followed by either hex'ad' or hex'bd'.

```
def a-[ = [hex]'ad'
def a-] = [hex]'bd'
```

In order to have the PC display square brackets the ASCII to EBCDIC conversion table which is called by the de5250 program will have to be modified in order to have the hex values 'ad' and 'bd' displayed as square brackets. (Normal 5250 emulation would modify these values to some other displayable characters.)

To modify this table use the config program and take the "select display emulation options" from the advanced options menu. Then customize the EBCDIC to ASCII table by keying 'ad' in the EBCDIC value and pressing Enter then changing the ASCII value to '5b' and pressing Enter. Similarly map the EBCDIC 'bd' to ASCII '5d'.

If Workstation Support is used with either a PC or PS/2 then similar changes are required to allow the use of square brackets.

With Workstation Support the user must update the keyboard profile to map the PC's square bracket keys to valid EBCDIC values and the user must update the Session Profile to allow the EBCDIC to ASCII conversion to display these characters. This can be done either directly with the CFGWSF program or under menu control of the PC Organizer. The keyboard mapping with Workstation Support allows the direct mapping of the PC square bracket keys to the hexadecimal values required for square brackets.

2.3 Trigraphs

There is an SAA extension available on the AS/400 which allows the C Compiler to accept a combination of characters instead of some of the special graphics. This will allow the C programmer to enter and create C programs with a more standard character set. This may cause the source program to become awkward to read but does allow any keyboard to be used to enter and display a source C program.

An example of the coding required is to place ??(in any location that requires a [and ??) to represent]. For a complete list of trigraphs see the *SAA CPI C Reference* manual.

An example of a C statement coded with trigraphs on a terminal which does not display square brackets and then on a terminal with the complete C character set is:

```
printf("%s",*cp??(-1??));
printf("%s",*cp[-1]);
```

2.4 Source Conversion from Other Systems

When transferring a document from an ASCII-based system to either VM or the AS/400 which uses EDCBIC it is very likely that the translation will not convert all the graphics as required for the C Language. The user should check the conversion table for these special characters.

Many displays including 5250s, and some 3270s do not display the C graphics as they are required for use in source program coding. In some cases if the file is viewed in update mode on these devices the special characters may be lost.

2.5 Distinct Upper and Lower Case Names

The C language treats uppercase and lowercase letters as distinct characters when used for any names except external program names.

For example a C function call to "Squeeze" will NOT use the function "squeeze", and a program reference to a variable "cnt36" will not interfere with the variable "cnT36", but a call to the program "Vbar" will call the program "VBAR".

2.6 Character vs Record Processing

In many C implementations the I/O operations are character by character. The AS/400 works on a record-by-record basis. This difference is sometimes characterized as the amount of buffered data between the terminal display and the program. Some of the symptoms of this difference may be in the lack of a capability of program verification of data entry as each keystroke is entered. The AS/400 will interact with the program on a full-screen or record-by-record basis. In addition to terminal differences the treatment of stream mode data is different. More is written about this difference under the stream mode data heading in this book.

2.7 Secured Pointer Usage

The use of a pointer variable on the AS/400 is controlled by the hardware and is slightly different from other C platforms. C/400 uses the 16-byte pointer that is implemented by the AS/400 hardware. Pointer arithmetic with this larger-than-normal variable works the same in C/400 as in any other C as long as the system is aware that this variable is an address pointer. Normal arithmetic on this pointer gives correct results. This includes pre- and post-incrementing or shifting by factors of other variables. However if a program attempts to do a direct move of some value into a pointer variable the flag bits which indicate that this is a pointer variable are lost and the system will not allow this variable to be used as a pointer. In other words if the application had been set up to move a pointer's value to another variable for some type of manipulation and then back to the pointer variable the hardware will detect this and not allow its use as a pointer.

2.8 User-Defined Data Stream

The terminal I/O generated by an AS/400 program may come from the regular ANSI C I/O routines, from an externally described display file, or from a user-defined data stream. The first two are explained in more detail later in this book. The following section explains where the user-defined data stream may be required.

The AS/400 normally works with 5250 type devices and thus uses a 5250 data stream. When writing any AS/400 program the expected data to be fed to the workstation handler is this data stream. C/400 is no exception to this and will generate 5250 data from the C output routines. For example printf will generate a 5250 data stream. This will be true on an AS/400 for any type of device actually connected to the system whether it is a 5250 device (local or remote), a PC running emulation or APPC communications, or an ASCII device on the

ASCII workstation controller. Some C applications on systems other than the AS/400 have been set up to generate a "generic" data stream and then pass this through a device-dependant routine before actual I/O. This technique allows the development of a common application that may be ported across different platforms. If an application written with this approach is to be ported to an AS/400 the user would have to write the routines to take this "generic" data stream and create a user-defined 5250 data stream.

2.9 Floating Point Implementation

The floating point implementation on the AS/400 and the PC uses the IEEE-488 format which is not the same as the C/370 implementation. The difference is in the definition of the length of the exponent and mantissa and the method of normalization of the value. It is possible to have different results on the AS/400 from those obtained from a S/370 C program.

2.10 Return Codes in C/400 Programs

Almost all functions defined in C/400 libraries will return a code to the caller after execution of the function. When a user defines a function in the program, a return code can also be specified. The return code can be used to control the flow of the program.

There can be many values returned from a function, and they can generally be grouped into the following categories:

- Return code to indicate true or false condition

  We can use a function to test whether a condition is true or false. For example, we can test whether a character c is a number (0 to 9) by using isdigit(c) function. If c is a number between 0 to 9, a positive number will be returned; otherwise, a zero will be returned.

- Return code to indicate result of an operation

  We can call a function to perform a certain operation. If the function request is successful, one kind of value will be returned, if not, another kind of value will be returned. For example, when fopen() is used to open a file for read, if the file open is successful, a file pointer will be returned to the caller, but if the file open fails, a NULL pointer will be returned.

- Return data

  A function call may return valid data. For example: We can use toupper() to convert a character from lowercase to uppercase. The value returned from toupper() is the uppercase of the original character.

- Where to find return code information?

  A description of the return codes from C functions can be found in the SAA publication *Common Program Interface C Reference* In the description of the each function, there is a short paragraph describing the return code.

The following example uses the three type of return codes and values previously discussed.

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
FILE *file1;
FILE *file2;
int i;
int j;
char c;
char s[10];

/* ----- test true or false condition ---------- */

printf("Enter any character\n");
c=getchar();
i=isdigit(c);
printf("Value of isdigit(c) is %d\n",i);

/* program to control flow based on return value */

if(isdigit(c))
 {
  printf("Yes, c represent a number\n");
  }
  else
   printf("No, c does not represent a number\n");

/* ----- use fopen for testing. file ZZZZ does not exist in system  */
/*       AAAA does exist in system */

printf("\n");
file1=fopen("AAAA","w+");
fclose(file1);
printf("Now open both file AAAA and ZZZZ for read.\n");
printf("Note that AAAA exist while ZZZZ does not exist.\n");
printf("\n");
file1=fopen("AAAA","r");
file2=fopen("ZZZZ","r");
printf("Value of file1 on AAAA is %d\n",file1);
printf("Value of file2 on ZZZZ is %d\n",file2);
printf("\n");
fclose(file1);
fclose(file2);

/* ----- convert character ----------- */

printf("Now will use toupper() to covert z to upper case\n");

printf("Upper case of character z is %c\n",toupper('z'));
 }
```

3. AS/400 C Environment

3.1 Environment Introduction

All HLL programs run in some environment. The AS/400 environment you are accustomed to with high level languages such as RPG and COBOL is defined below the machine interface and includes the Process Access Group (PAG). The PAG is a collection of storage areas containing information required for your job to execute. The PAG can be paged in and out of memory with one disk access rather than being written a page at a time. The PAG has separate areas for control information, program variables, and open data paths for database and display files. The actual executable code is stored outside of the PAG. Everything that your program needs in order to execute is defined in the PAG.

With languages such as C and Pascal, support is required for multiple entry points as well as external variables. In RPG and COBOL programs there is only one entry point; the name of the program is the name of the entry point. However, with languages such as C and Pascal, the system needs to know the name of the entry point and in what program the entry point can be found. In addition to this, external data needs to be supported, and a debug facility that can handle external as well as internal variables is required. This necessitates a different type of program model and environment. The new program model for multiple-entry-point languages like C and Pascal is called the Extended Program Model (EPM) and is defined above the machine interface standard program model. The EPM includes a run time environment which supports the unique requirements of EPM languages like C and Pascal.

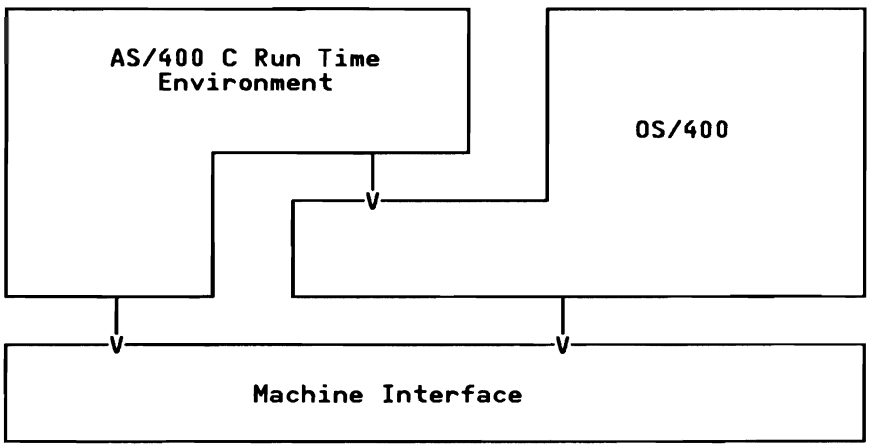


Figure 3-1: AS/400 EPM Environment

External Variables

External variables are not supported in the standard AS/400 program model. An external variable is one accessible to multiple programs. In contrast, a local variable is visible only within a program and exists only as long as the program is active. Variables in RPG and COBOL are local; each program invocation causes a new set of the variables to be initialized. If you wish to allow a second program to use your variables, they must be passed as parameters on the program call or written to a data area of some type and read by the called program.

External variables are available to multiple programs and exist from one program invocation to another. In C you specify a variable as being external by where it is defined. For example, in the following program, the integer (int) variable "value1" would be external since it is specified outside of the body of the C program or before the "main()" statement.

```
/* this is a C program example of external data */
int value1;
main()
{
  /* program statements would be coded here */
}
```

If the variable "value1" were specified after the statement "main()" it would be a local variable. It would only be visible within this program and only exist as long as this program is active.

```
/* this is a C program example of internal data */
main()
{
  int value1;
  /* program statements would be coded here */
}
```

External variables are defined and need to be supported across compilation units (programs, subprograms or functions). Local variables are those which exist only within a compilation unit. They must be scoped (restricted) to a compilation unit so that variables of the same name in different compilation units can't destroy each other. For example, you could have two programs or subprograms that both have a variable called x defined as a local variable. These would be two distinct variables, each only visible within its own program.

Multiple Entry Points

As mentioned earlier, C programs can have multiple entry points. If you call a C program, the default entry point is the routine or function called main. To call any other entry point, you must specify the name of the entry point you wish. The following is an example of a C program with multiple entry points.

```
/* this is a C program with multiple entry points */
main()
{
  /* program statements for main function   */
}
verify()
{
  /* program statements for verify function */
}
error()
{
  /* program statements for error function  */
}
```

The entry points in the previous program are "main", "verify" and "error". Another program could call any one of these entry points or functions. When you call the program by name you default to the entry point "main". You will note that the program name is not in the actual code; this works the same as RPG or COBOL where the compiled program name defaults to the name of the source file member which is being compiled. The program name and library name can, of course, be overridden on the CRTCPGM command. To display which source file and member name was used to create a program, you can use the DSPOBJD (Display Object Description) command and specify "display service attributes".

3.2 Extended Program Model

The AS/400 program model for multiple entry point languages like C and Pascal is the Extended Program Model (EPM). The EPM contains the following components:

- The environment
- The debugger
- The exception handler
- The session manager.

Each of these sections will be addressed separately, starting with the EPM environment or the C run time environment.

3.3 EPM Environment

There are four major components of the EPM environment:

- Environment control block
- Automatic storage
- System heap
- User heap.

Each of these areas are separate allocations of memory. The environment control block has pointers to the three other areas. The automatic storage, also known as the user stack, contains the user variables. This is similar to the PASA (program automatic storage area) in the standard program model. The

system heap contains linkage information and run time storage as well as user static and static external variables. The user heap is storage controlled by the C program by using functions such as malloc (allocate memory), free, new, and dispose. The following is a schematic of how these areas tie together.

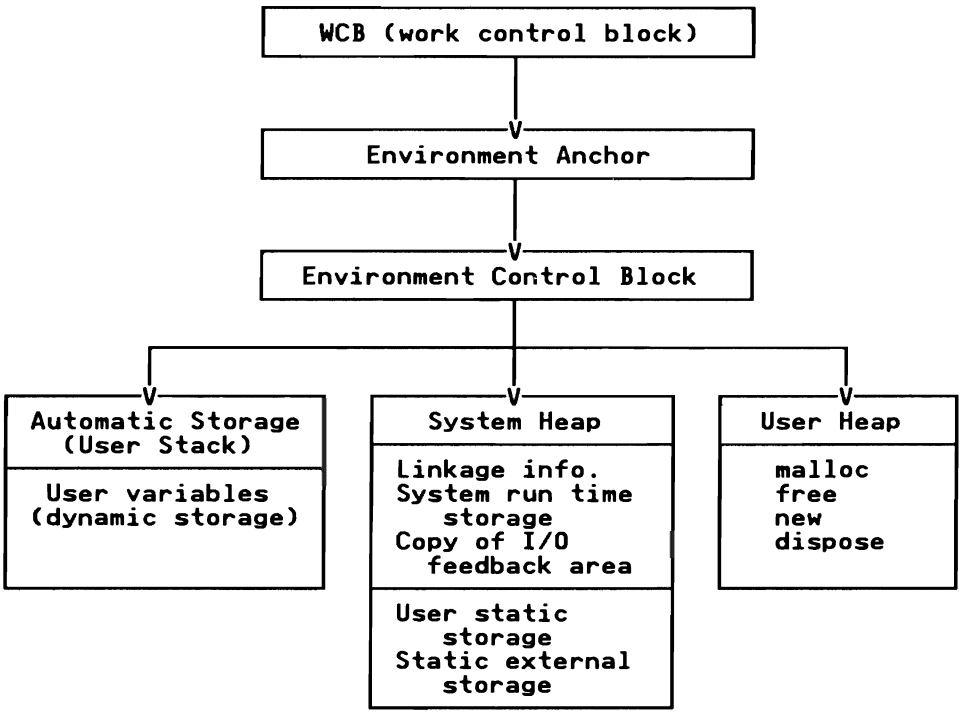


Figure 3-2: AS/400 EPM Environment Contents

By default these areas are not included in the PAG, since the PAG has a size limitation of 4 megabytes. The automatic storage and heap storage areas can each approach 16 megabytes in size, and this would not be possible if they were forced to be in the PAG. If the size restriction is not a problem, you can choose to put these areas in the PAG via the PFROPT parameter of the SETPGMINF command. You can also set the sizes of these areas via SETPGMINF. The approximate minimum sizes for these areas are 16K for the automatic storage and 32K each for the system and user heap. Once the size of these areas is set either by default or SETPGMINF, they are not trimmed like the PAG. If you do not put the EPM environment contents in the PAG, they will be unaffected by the PURGE-*YES or PURGE-*NO setting; they will be demand paged like any application code.

When you open database files in C, space is allocated in the system heap. This is similar to the open data path created in the PAG for the standard program model. The fopen routine allocates space in the system heap as well as a pointer to the PAG. When you pass a file pointer in C, it points to the space in the system heap. An interesting point to note here is that when you want multiple C programs to share a file which has previously been opened, there is no additional overhead. This is because you are passing a pointer rather than doing an open. In the standard program model, a second shared file open causes an abbreviated open which is less overhead than a full open, but still requires CPU.

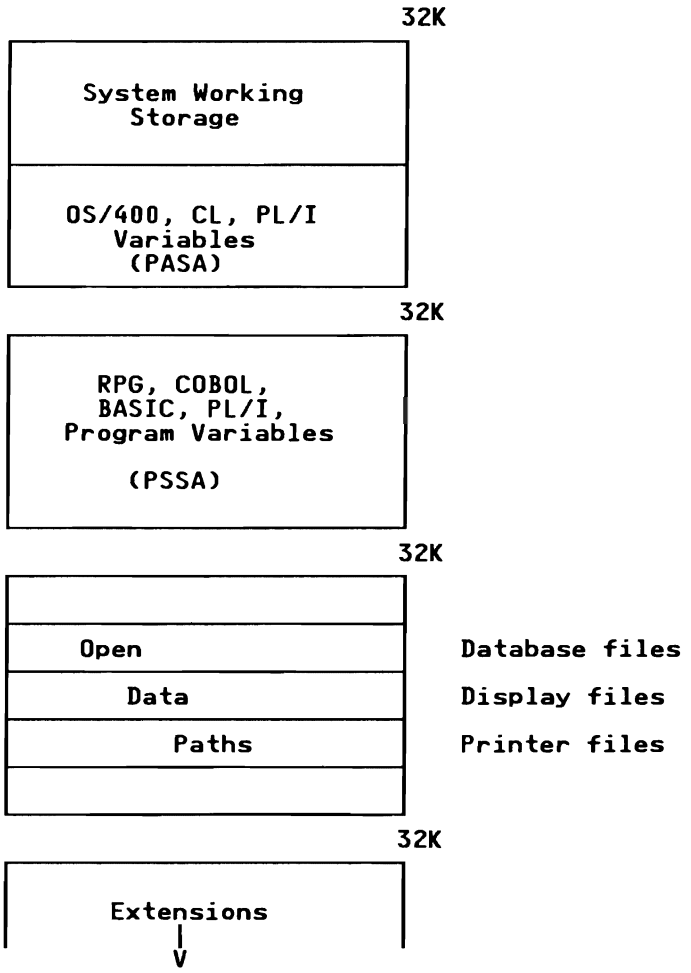


Figure 3-3: Process Access Group (PAG) for Standard Program Model

Even if you do not include the C areas in the PAG, there will still be a PAG. The PASA will contain some internal control routines, but the size of the PSSA (program static storage area) will be zero. By using the Display Program (DSPPGM) command you can see what the size of your C executable code is (program size in bytes) as well as the size of the automatic storage in the PAG. Note that this is not the same as the automatic storage area in the EPM model. The automatic storage, system heap and user heap of the EPM are in addition to the PAG and C executable code. Also, if your program has a terminal session, there will be a separate area allocated in memory for the session manager as well.

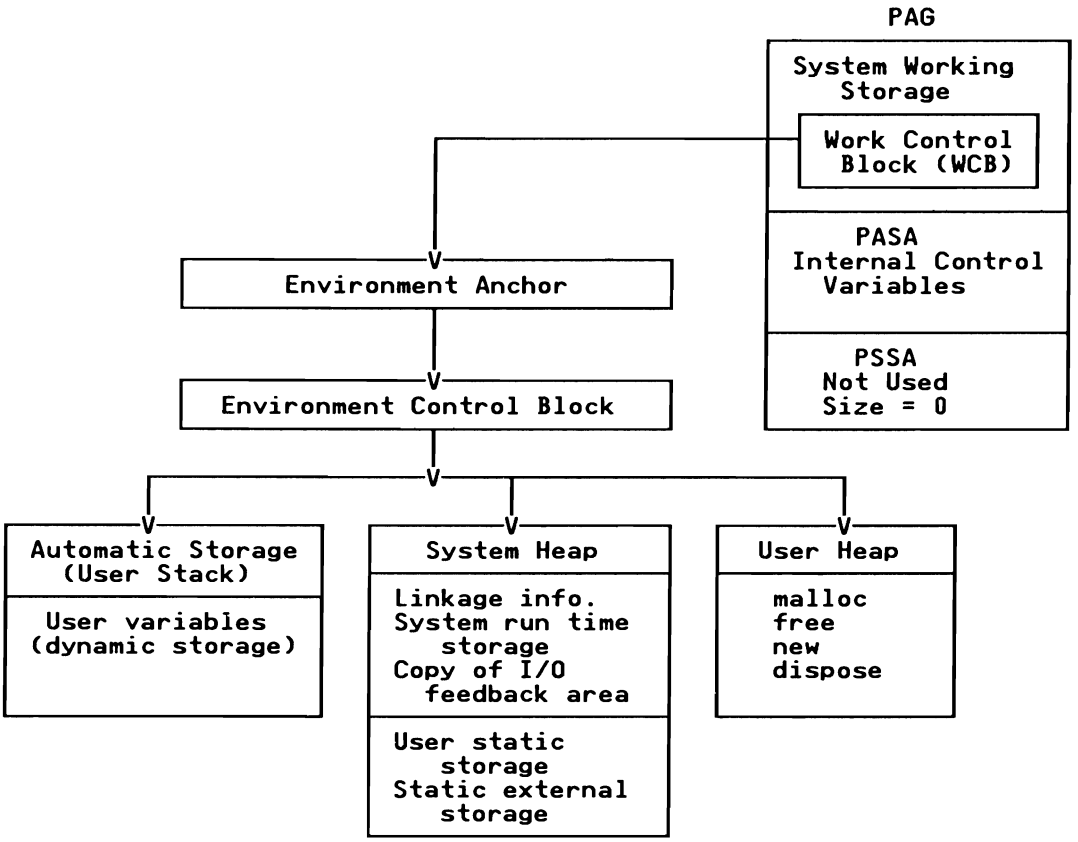


Figure 3-4: PAG for Extended Program Model

3.4 Debug

EPM debug is a special debug facility for EPM programs. It sits on top of the OS/400 debug. So if you issue the STRDBG command and call a C program, EPM debug intercepts your request and automatically puts you into EPM debug mode. The EPM debug functions quite differently from the OS/400 debug. In OS/400 debug, you would enter debug mode, set your break points and then call the program. For EPM debug you issue the STRDBG command and then call your program. Then EPM debug stops to allow you to set the break points, and you enter "go" to start or resume execution. You can key in "help" while in EPM debug to see the list of commands available to you. Aside from that, there is no other online EPM debug help available. However, Appendix A of the *C/400 User's Guide* does explain each command in detail as well as any error messages you might receive. When you have finished debugging your program, you can exit the program or enter the EPM command "end" or "quit" to end the EPM debug session. This will not end the OS/400 debug session, however, so you must issue the ENDDBG command to back all the way out of debug mode.

In order for the EPM debug to function, your program must be compiled with the OPTION parameter *DEBUG. The default option is *NODEBUG on the CRTCPGM command. If your program calls other programs, and you want to debug each program, all of the program objects must be compiled with the *DEBUG parameter specified. The SETPGMINF command also has a parameter

to set debug off or on; the default here is *ON. If you specify debug *OFF on the SETPGMINF command you will still be able to use OS/400 debug but not EPM debug. EPM language variable names and statement numbers are not available under OS/400 debug alone.

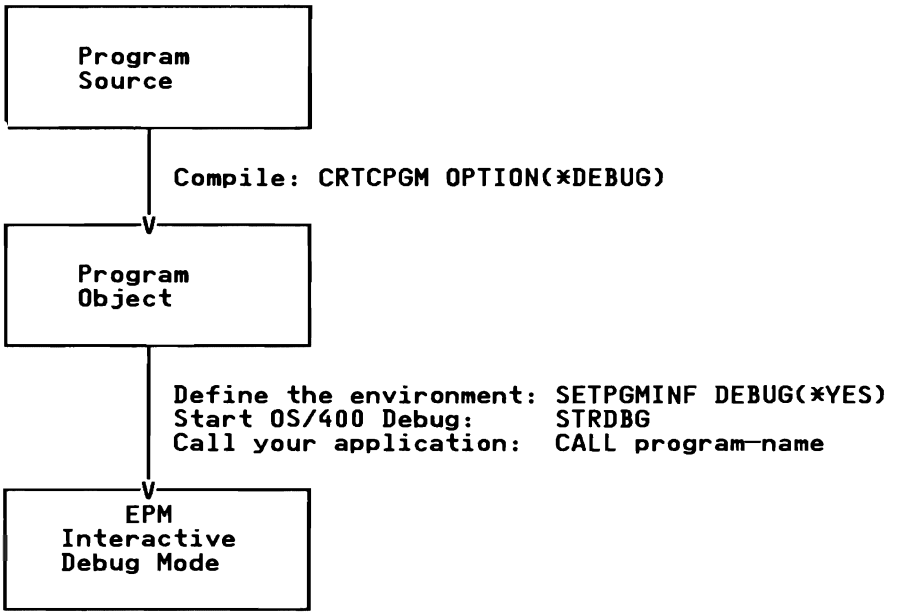


Figure 3-5: Debugging a C/400 Application

If you are using SQL in your C program the CRTSQLC command is issued to do the SQL pre-processing. This procedure in turn calls CRTCPGM, but does not allow the input of any optional parameters such as *DEBUG. If you wish to use debug in a C SQL program, you can either create your own CRTCPGM command with *DEBUG as the default and put it in a library ahead of the QCC library, or you can create your own CRTSQLC command. If you create your own CRTSQLC command you must specify *NOGEN for the CRTSQLC and then issue the CRTCPGM command using input from the physical file QSQLTEMP in the library QTEMP with a member name the same as your program name.

Once you have successfully tested and debugged your C programs, it would be a good idea to recompile them with the *NODEBUG option. This will result in smaller program sizes which may be critical in tight performance situations. If you can keep your program within a 64K segment size, branching within the program will be much faster than if multiple segments are involved.

3.5 Exception Handler

The EPM environment issues exceptions if something is wrong. The way your application would process these exceptions is dependent on the EPM language(s) that are involved. In Pascal you use the ONERROR procedure. In C you use the ANSI-defined signal handling mechanism. Both types of exception handling work in the EPM environment.

Handling EPM exceptions with a signal handler is comparable to monitoring for a message (MONMSG) in the standard program model. With MONMSG you

specify the message number you want to check for and intercept. With signal-handling you specify the name of the signal-handling routine or exception handler you wish to get control for various types of exceptions.

If you do not code an exception handler, then the EPM default actions will be taken when an exception occurs. The exception handling is based on the type of message issued by the operating system: ESCAPE, NOTIFY, or STATUS. ESCAPE will halt the program and issue an EPM diagnostic message, NOTIFY will send the default reply to the sender of the message, and STATUS will issue an EPM diagnostic message.

To handle the signals in your C program you would include the signal function by coding the following statements:

```
#include <signal.h>

void(*signal (int sig, void(*funct) (int))) (int);
```

This function determines how an exception received in the sig argument is treated. The value of sig can be one of the following:

- SIGABRT - Abnormal termination
- SIGFPE - Arithmetic error
- SIGILL - Illegal operation
- SIGINT - System interrupt
- SIGSEGV - Invalid storage address
- SIGTERM - Program termination request
- SIGUSR1 · Reserved for user signal handler
- SIGUSR2 - Reserved for user signal handler.

You can specify a user-defined handler routine or a predefined handler on the funct parameter. Initially all signal handlers are set to SIG_DFT which is the default signal handler. If you set the signal handler to SIG_IGN, that type of signal will be ignored. Another handler, SIG_ERR is defined in <signal.h> and it indicates that an error was returned from the signal function.

All C/400 routines in an environment share the same signal handlers. This means that signals are scoped to the EPM environment boundary, and an exception is externally visible at the environment boundary. However, the signal handler does not have to be in the same environment; you can pass a pointer to a signal handler which is in another EPM environment. Keep in mind that when a signal handler is called, its corresponding signal action is set to SIG_DFL. You must reset the signal action if you want to handle the same signal again. If you don't do this, the default action will be taken on subsequent exceptions. You can reset the signal action in the signal handler routine or back in your application code.

When you are coding a user-defined signal handling routine, you can get more information from the EPM exception handler by coding the sigdata function in your routine. This function returns a pointer to a structure of type sigdata_t. This structure contains pointers to three other structures: exmsg_t (for information about the exceptions), usrmsg (for issuing a diagnostic message to

the caller of the program), and sigact (for the default actions on entry to the signal handler).

For those who have used the technique of removing observability of programs to conserve disk space on the AS/400, be aware that you cannot remove observability of C or Pascal programs. Observability is one of the three tables required for exception handling at execution time in the EPM environment. If you attempt to remove observability of a C program via the CHGPGM command you will receive a message saying "cannot remove observable information".

Aside from EPM environment exceptions, you will need to handle errors that occur when you are using EPM and C library functions. EPM library routines issue exceptions, where C library functions only set a return code and sometimes a value in the global "errno". Consult the *C/400 User's Guide* for information on what the error handling method should be for a specific library routine or function. If "errno" is set, you can check to see if it is equal to "EIOERROR" or "EIORECERR". If it is equal to one of these, a CPF exception has occurred and you can check the global"_EXCPDATA" for more information. There are only two values of "errno" that are defined in ANSI C. C/400 will issue more than these two values, which is helpful, but since this is an AS/400 extension to C, it can be a factor in the portability of the code you generate.

3.6 Session Manager

Terminal input and output is managed by the terminal session manager. The EPM languages share the session manager, and there is only one session manager regardless of how many environments are active. The terminal session manager supports the following:

- Scrolling backwards and forwards
- Command retrieval using F9
- EOF signaling using F13
- Extension of the input line with F21
- Output of characters below the hexadecimal character X'40'.

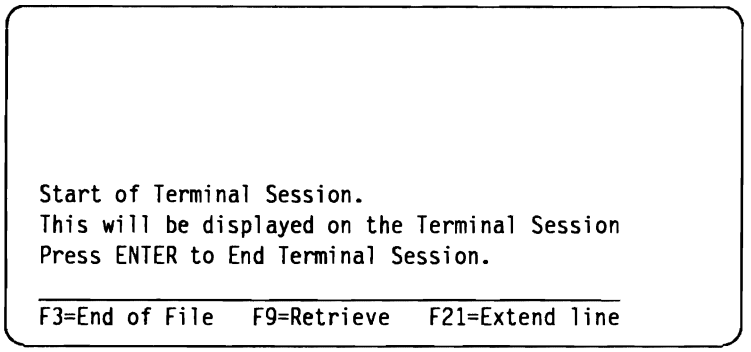
If you output characters below X'40', they will be treated as 5250 terminal control characters.

Input and output operations performed on a stream result in I/O operations to the terminal when the standard C/400 files stdin, stdout and stderr are used. If the job is not interactive and stdout and stderr are specified, the C/400 compiler overrides the printer file to QSYSPRT and the information that would normally be displayed is printed or spooled for printing. If stdin is specified and the job is not interactive, the file QINLINE is used to retrieve the input data.

If the following program were run:

```
/* program to display on terminal */
#include <stdio.h>
main()
{
printf("This will be displayed on the Terminal Session\n");
}
```

this is what would appear on the screen:



When our program finished, the session manager sent an implicit read to the terminal. This was to ensure that the data we sent to the terminal remains on the display rather than flashing briefly and then disappearing. From this display you can scroll through the terminal session. If this program had put out ten screens of data, you could roll back and forth and view the ten screens of information on the terminal. This information is stored in a separate area in memory for the terminal session. The size allocated for this terminal session can be specified on the SETPGMINF command by using the SSNATTR (session attributes) parameter. You can specify a session size between 8K and 16 megabytes; the default size is 32,000. Care should be taken to keep this size reasonable so that performance is not adversely affected. You can also specify a buffer size for the session between 80 and 255 bytes, with the default being 160. The session manager does not clear the display buffers until your job ends or you issue the command RCLRSC (Reclaim Resource).

3.7 EPM Application Library

The EPM application library is a set of programs and routines which are included with the AS/400 C compiler. They can be categorized as:

- ınterface programs
- Conversion routines
- Data area access routines
- File interface routines.

The interface programs are provided for inter-language calls. You use the QPXXCALL function to call an EPM language program from another program. You can also use QPXXCALL to create a user- controlled environment between EPM language programs. The QPXDLTE program is used to delete the user controlled environment.

When a C program is called, the environment for the program is created. When the C program finishes and returns to the calling program, the environment is torn down. This requires a certain amount of system resource. When you would be calling the same C program or set of routines many times it may be to your advantage to have these programs or routines in an environment that would not be destroyed on every return. The way to do this is to create your own user-controlled environment. A user-controlled environment exists until you delete it or you sign off. One word of caution here is that the second time

you call a program in the user-controlled environment the variables are NOT re-initialized; they are as you left them after the previous call.

Be aware that all the parameters used for the QPXXCALL program are fixed length. The program name is 100 characters long and is case sensitive. So if you want to call a C program from an RPG or COBOL program you would have to declare a variable of 100 characters and move the program name or entry point name into this variable. Since there are other parameters after the program name, the QPXXCALL routine will be looking in position 101 for the second parameter. All parameters will have to be defined for their full length and initialized appropriately.

You would use the EPM conversion routines to convert packed or zoned decimal numbers to other data types, or to convert other data types to zoned or packed decimal numbers. The conversion routines include the following:

QXXDTOP - Floating point to packed decimal

QXXDTOZ - Double to zoned decimal

QXXITOP - Integer to packed decimal

QXXITOZ - Integer to zoned decimal

QXXPTOI - Packed decimal to integer

QXXPTOD - Packed decimal to double

QXXZTOD - Zoned decimal to double

QXXZTOI - Zoned decimal to integer.

The data area access routines are used to pass information to or from your program. The two routines are QXXRTVDTAA (retrieve data area) and QXXCHGDTAA (change data area). These routines are well documented in the *C/400 User's Guide*.

The file interface routines provide a way to use some of the system functions. Among these functions are the following:

- Commitment control

  QXXCOMMIT - start commitment control

  QXXROLLBCK - roll back changes

- Set record format name

  QXXFORMAT

- Work with device files

  QXXACQUIRE - acquire program device

  QXXRELEASE - release program device

  QXXPGMDEV - set default program device

- Set separate indicator area

  QXXSINDARA

- Obtain feedback information

  QXXIOFBK - obtain I/O feedback information

QXXOPNFBK - obtain open feedback information

QXXDEVATR - obtain device attributes feedback information.

The commitment control functions are self-explanatory. The set record format name allows you to specify the record format name to be used for subsequent I/O operations. The device file functions allow dynamic allocation and de-allocation of devices. The set separate indicator area allows you to specify that the indicators not be part of the I/O buffer, but be contained in a separate area. The feedback information functions allow you to access feedback information; the structure of the feedback areas is documented in the *C/400 User's Guide*.

3.8 SETPGMINF

The SETPGMINF command defines what is to be included in the application environment, the memory sizes of the various components, whether the EPM should be in the PAG, what the error threshold is, and if debug should be on or off based on the parameters you supply.

On the ROOTPGM, SUBPGM, and LIBFILE parameters you identify the programs, subprograms and linkage information required. ROOTPGM is an EPM program that contains the information required to establish the application environment. SUBPGM specifies a list of program objects that should be included in the environment. LIBFILE is a library information file that is used to resolve any outstanding external references, after all of the linkage information supplied by the ROOTPGM and SUBPGM parameters has been processed. PFROPT indicates whether the EPM environment should be included in the PAG. Using the HEAPSIZE, STACKSIZE, AND SSNATTR parameters you can specify an initial size for the system and user heap, automatic storage, and the terminal session. With the parameter RUNATTR you can limit the number of non-fatal errors that will be allowed before processing will be ended. Finally, you can indicate whether you want EPM debug to be on or off when OS/400 debug is active. If you turn EPM debug off, you can still use OS/400 debug to debug your programs, but EPM language variable names and statement numbers are not available under OS/400 debug.

3.9 EXTPGMINF

The Extract Program Information (EXTPGMINF) command extracts external linkage information from the EPM programs identified on the PGM parameter. External linkage information includes external variables and entry points that are used by the SETPGMINF command at run time. External linkage information can only be extracted from EPM programs. This linkage information is stored in a library information file. This command allows you to create a file to store names of affected entry points, instead of specifying each program name on the SUBPGM parameter of the SETPGMINF command. This will be especially useful with large complex applications.

When EXTPGMINF is used, a date/time stamp is included with the data that is written to the library information file. When this library file information is subsequently used, the date/time stamp is compared to the date/time stamp of the program containing the entry point(s). If the date/time stamps are different,

the program has been changed since the EXTPGMINF was done, and the extract needs be re-done to update the library information file.

3.10 Application Performance Tuning Aid

The Application Performance Tuning Aid (APTA) is a system utility that allows you to take multiple EPM programs and subprograms and bind them together into one single program object. This utility is not included with the C compiler, but is available separately as a PRPQ (Program Request Price Quotation). Any EPM programs can be bound; this includes Pascal as well as C. Binding on the AS/400 is the process of combining two or more program objects that make frequent external calls to each other. This may improve the performance of your application, depending on what you choose to bind together and in what sequence.

In order to bind a C program, it must have been compiled with the option GENOPT(*ALWBIND) specified on the CRTCPGM command. When combining a number of programs, you cannot bind more than one that has a default entry point. In C this would be a program with a procedure called "main" You cannot bind programs which contain SQL statements, and you cannot bind C library functions into your programs.

Normally, the programs that will perform better when bound are the ones that are used together frequently. At program run time, each call to a separately compiled program causes an external call sequence to be generated. External calls are expensive because they require a great deal of system overhead. When a set of programs is bound together, any external references that are made between those programs are resolved, and any external calls between them become internal calls in the bound program.

To help you determine which programs are candidates to be bound together, you can use the Trace Job (TRCJOB) function of the AS/400. Since TRCJOB can produce a listing of the calls your application makes, you can use this to determine which programs call each other repeatedly. Once you have made a selection of candidates to be bound, it would be a good idea to check the size of the programs using the DSPPGM command to determine if your bound program will exceed the 64K segment size. If you exceed this size your program will be stored as multiple segments, and branching between segments is more costly than branching within a segment. A bound program that is too large may not perform any better than the unbound individual program objects.

At this time there is no system aid that recommends what to bind together or in what sequence the programs should be bound. Keep in mind that if two programs call each other frequently they should be bound next to each other, not with a third program intervening. Also, it may be more advantageous to have multiple smaller bound programs being called from an application than to have all of these called programs bound together into one large program. Since no changes are required to your source files when binding programs together, you can take a "trial and error" approach. You can bind the best candidates together and see if this improves performance. If it does not, then you can try another combination.

Create Bound Program (CRTBNDPGM) is the command used to bind programs together. If you wish to bind multiple programs together it is not necessary to

bind them all in one step. You can bind two together and specify GENOPT(*ALWBIND) on the CRTBNDPGM command. This will allow you to later bind this composite program with other bindable programs. If you bind programs in multiple steps like this, you can test at each step to see if your last combination of bound programs truly did improve performance. When you have reached a final stage and have bound together what appears to be the ultimate combination of programs, you should specify GENOPT(*NOALWBIND) on the CRTBNDPGM command. This does not undo the original binding, but merely removes the information necessary for binding the composite program again. Since this information would be stored with the final program object, GENOPT(*NOALWBIND) reduces the amount of storage required.

3.11 Location of C/400 Runtime Routines

The OS/400 routines required to run C/400 programs are not included in the same library as the C compiler(QCC). For version 1 release 2.0 these routines are ordered and installed as program product 5799-XAY. The restore licensed program (RSTLICPGM) procedure will load a separate library QCSYS with the required runtime routines. During the install process the system library list will be modified by adding the QCSYS library ahead of the QSYS library. In later releases this process will not be necessary since the routines in QCSYS will be included in the QSYS library.

4. Data Manipulation

4.1 Alignment of Data

Fields in an AS/400 database file do not have special rules for data alignment. That is, subsequent fields in a record follow immediately behind the preceding fields. The length of an AS/400 data base record can easily be calculated by adding the length of individual fields together. Furthermore, the length of numeric fields can be specified at the time when the file is created. For example, we can define a field in a record to be seven-digit zoned decimal and we can define another field of three-digit zoned decimal in the same record when we create the file. Fields of same numeric data type, can be defined with different length.

In C/400, the user cannot specify the length of a numeric field. The user can only specify the type of numeric field and there is a fixed length of storage space required by each numeric data type. Currently there are seven types of numeric data that can be defined. They are: integer, double, float, long integer, signed integer, short integer and unsigned integer. Numeric data under C/400 has an alignment requirement. Because of this property, calculating the length of a C database record is not so straightforward, specially where there is a combination of alphanumeric fields and numeric fields.

The following examples use integer and alphanumeric fields to illustrate the effect due to data alignment requirement of C/400. Integers always align at a word boundary (multiple of 4 bytes).

Let FLD1 be an alphanumeric field of 5 bytes, FLD2 be an alphanumeric field of 30 bytes and FLDNUM be an integer which will occupy 4 bytes.

The length of a record made up of FLD1,FLD2 will be 35 bytes (5 + 30 = 35).

The length of a record made up of FLD1,FLD2,FLDNUM will be 40 bytes. FLD1 and FLD2 together take 35 bytes, but since it does not end at a word boundary (multiple of 4 byte), the FLDNUM can only start at position 37. Therefore the record length is (5 + 30 + 1 +4) = 40.

A Method to Determine Record Length

Usually the greatest need to know the length of a record is when we want to create a new file, otherwise, we can use the function sizeof() to store the length of the record in a variable inside the program. An easy way to find out the record length of a record is to use a short program to do the calculation.

The usual way to define a record buffer in a C program is to define a structure. The following is an example of a simple program to find the record length:

Sample Program to Find the Record Length of a Structure

```
 struct record {
  char fld1 [5];
  char fld2 [30];
  int fldnum;
  float fldnum1;
    } record ;

main()
{
 printf("length of record is %d\n",sizeof(record));
}
```

The length of this structure is found by using the sizeof() function. Once the record length is known, this value can be used to create a physical file on AS/400 with CRTPF command.

Referencing Numeric Elements in Structures

There can be various type of numeric data elements in a structure. As long as the correct elements are referenced, C/400 will automatically do the data alignment to ensure data is placed in the correct position. For example, if we have two structures·

Sample Structures

```
struct a {
   char fld1a[7];
   char fld2a[15];
   int fldn1a;
   float fldn2a;
  } a;

struct b {
   char fld1b[10]
   int fldn2b;
   char fld2b[40];
 } b;
```

If we need to set fldn1a in structure a equal to fldn2b in structure b, we would code:

```
a.fldn1a = b.fldn2b ;
```

Processing an AS/400 Externally Described File with C/400

Since AS/400 externally described data files do not have special data alignment requirements, when a C/400 program accesses an AS/400 database file, the database layout and the file buffer defined in the C program (usually in the form of a structure) may not match. However, since the most common numeric data types used in AS/400 database are packed and zoned fields, and under C/400 there is no data type of packed or zoned, such a conflict does not appear.

In order to use a zoned or packed field from an AS/400 database file, the field must first be read into the C/400 program as an alphanumeric field. The field will then be converted to an integer or floating number by functions like QXXZTOI,QXXPTOI. Similarly, when writing to an AS/400 database file, the integer or floating number must first be converted to a character string by using functions like QXXITOZ, QXXITOP. A detailed description of QXXITOZ etc. can be found in "Appendix B - The EPM Application Library" of the *C/400 User's Guide*.

Use of a Logical File to Map Fields

AS/400 logical files support field mapping, that is, the attribute of a field can be overridden by a logical file. For a more detailed description, please refer to the *AS/400 DDS Reference Manual*. The following is an example of a logical file with field mapping and the physical file it is based on.

Example of Physical File and Logical File with Field Mapping

Physical File - PHYSAM

```
     R FORMAT1 TEXT('FILE PHYSAM')
       FLD1          8
       FLD2          8S  2      TEXT('ZONED DECIMAL')
       FLD3          8
       FLD4          7   2      TEXT('PACKED DECIMAL')
       FLD5          8
       FLD6          4B  0      TEXT('BINARY NUMBER')
       FLD7          8
       FLD8          8F  2      TEXT('FLOATING NUMBER')
```

Logical File - LOGSAM

```
     R FORMAT1                   PFILE(PHYSAM)
       FLD1
       FLD2          F
       FLD3
       FLD4          F
       FLD5
       FLD6          F
       FLD7
       FLD8
     K FLD1
```

A C/400 program can use this logical file definition by defining a structure. Numeric fields in physical file will be read or written as floating point numbers through the logical file:

A C/400 Structure that can Read Records from LOGSAM

```
struct record {
    char fld1[8];
    float fld2;
    char fld3[8];
    float fld4;
    char fld5[8];
    float fld6;
    char fld7[8];
    float fld8;
    } record;       /* size of logical record is 48 bytes */
```

This is a particular case where by mapping fields through logical file, numeric data can be read directly into a C/400 structure for processing. The most important thing about this sample PHYSAM is that all numeric fields are already at word-boundary level (multiple of 4 bytes), so when a C/400 structure is declared, there is no additional space between alphanumeric field (such as fld1, fld3 ...) and numeric field (such as fld2, fld4 ...) due to data alignment effect.

If the alphanumeric fields in physical file do not end on a word boundary (multiple of 4 bytes), the C/400 structure will not match the physical file. For example:

Example of Physical File, Alpha Fields not End at Word Boundary

Physical File - PHYSAM1

```
        R FORMAT1 TEXT('FILE PHYSAM1')
          FLD1              5
          FLD2              8S  2       TEXT('ZONED DECIMAL')
          FLD3              5
          FLD4              8   2       TEXT('PACKED DECIMAL')
          FLD5              5
          FLD6              8B  0       TEXT('BINARY NUMBER')
          FLD7              5
          FLD8              8F  0       TEXT('FLOATING NUMBER')


Logical File - LOGSAM1

        R FORMAT1                        PFILE(PHYSAM1)
          FLD1
          FLD2               F
          FLD3
          FLD4               F
          FLD5
          FLD6               F
          FLD7
          FLD8
        K FLD1
```

The structure defined in C/400 program to read LOGSAM1:

A C/400 Structure that can Read Records from LOGSAM1

```
  struct record1 {
      char fld1[5];
      float fld2;
      char fld3[5];
      float fld4;
      char fld5[5];
      float fld6;
      char fld7[5];
      float fld8;
      } record1;      /* size of record1 is also 48 byte */
```

In record1, due to word-boundary alignment requirement, three spaces will be left after each alphanumeric field, making a total length of the record of 48. When the C/400 program reads the logical file LOGSAM1 with the structure record1, data in the record buffer will be incorrect.

Therefore, field mapping through logical files may help to avoid using data conversion routines, but it depends on the layout of the physical file.

4.2 Stream Mode Data

Referring to Chapter 4 of *C/400 User Guide*, ANSI C defines a stream as a sequence of data that is conceptually read and written a character at a time and it has a close relationship with file. File is a source of data and stream is a medium that channels data from file to application. So file is similar to a reservoir and stream is similar to a hose through which we can get water.

One characteristic that can be derived from this definition is that stream is device independent. It can be mapped to different kinds of devices. The standard input/output device which will be mapped to the stream is the display terminal, but we can also map the stream to disk files, printers, tapes etc. Such a concept is not new to AS/400 developer. In AS/400, when we use OVRTAPF, we can override a device file (in this case the tape file) to a database file, then we use the CPYF command to copy data to or from another database. In this case, we are redirecting the input/output, originally destined to a database file, to a device file.

One significant difference between C/400 and other C platforms (for example under UNIX) is that in C/400, all input/output is fully buffered. That is, logically inside a C/400 program, data can be handled character by character (or byte by byte), but physical I/O is always done at a record basis. AS/400 will always present a record to the C/400 program and write a record to the file instead of character by character, regardless of whether the C/400 program is working or a text file or a binary file. Because of these AS/400 and C/400 characteristics, there can be a significant difference in some operations between C/400 and other C platforms. For example, under PC C or UNIX C, by using unbuffered stream, the application can respond immediately to a keystroke pressed by the operator from a display (and without pressing the Enter key), but with C/400, since all input/output is fully buffered, after pressing that particular key from the display terminal, the data is still in the buffer and the operator must press Enter key in order to send the data to the application.

Therefore, for users who plan to convert C applications to C/400, they should keep in mind that there could be a difference in some operation steps, especially relating to the effect of fully buffered I/O.

4.3 Text Stream Mapping on AS/400 to Source File

C/400 handles two kinds of streams - text stream and binary stream. A detailed description of text stream and binary stream can be found in Chapter 4 of *C/400 Users Guide*.

It is important to be aware of the difference between C/400 text stream on AS/400 with text stream on other C. For simplicity of discussion, we will use text file as an example. The text file created by C/400 is a physical file with fixed record length. The default record length of the text file is 266 byte and record length can be specified through lrecl keyword in the fopen statement. The text file can be dynamically created and it is equivalent to an AS/400 Source file. The first 12 bytes are not used and will be left blank. When the file is opened as

a text file, data written will always start at the 13th byte of the record. One record is read or written at a time, although logically in the C/400 programs, data is handled in a character mode.

Example of Writing Characters to a Text File

The following is an example of writing characters to a text file. Note that no record length is specified and therefore the record length will take the default value of 266 byte. Also note that a special character is added at end of the file. This is to ensure when reading the text file character by character, the read program knows when it reaches the end of the text file (note that in reading a text file, data will be read through fgetc() and is character by character, therefore feof() function will not work in this case).

```
Writing to a Text File:

main()
 {
 FILE *file1;
 char c;

  file1=fopen("TEXTFILE","w+");

  while((c=getchar())!='@')  /* '@' represent end entry   */
  {
    if(c!='@')
     {
      fputc(c,file1);
     }
   }

      fputc('\0',file1);  /* HEX 00 ('\0') means end of text in file */

      fclose(file1);

  }
```

This program will create a text file in library QTEMP since no library has been specified in the program. When DSPFD is used to display the description of the file, TEXTFILE is a physical file with record length of 266. When use DSPPFM to display content, we can see that the first 12 bytes of the record are left blank (filled with HEX 40). The fields starting from 91th byte are filled with spaces (HEX 40). Only 78 characters are entered per line because only 78 characters can be entered from the screen each time (assuming that F-21 is not used to extend the line).

The maximum data that can be stored in a C/400 text file record is (266- 12) = 254 bytes. C/400 will write a record to the text file whenever a carriage return character ('\n') is encountered. When data is entered from screen, since a maximum of 160 characters can be entered each time from the input line of the screen (by pressing F-21 to extend line), only 160 bytes out of 266 bytes in the

text file record is used to store data. The new line character '\n' will not be written to the text file.

In reading the text file, C/400 will truncate all the trailing spaces in the record and automatically insert a new line character '\n' to the input. All leading spaces in the record will also be ignored.

The following example illustrates how the C/400 will write records to a text file. In this example, the input file is from TEXTFILE and the output file is TEXTFILY. Characters are read from TEXTFILE and then written to TEXTFILY. But instead of writing exactly every character read from input file, a new line character '\n' is written to TEXTFILY after every 10 characters from TEXTFILE is written. The result is that instead of having an output file exactly the same as TEXTFILE, the TEXTFILY contains many more records and each record contains 10 characters only. This shows that whenever C/400 encounters a new line character '\n', it will write a record to the text file.

Writing to a Text File with New Line Character '\n'

```
main()
 {
 FILE *file1;
 FILE *file2;

 char c;
 int j;

 file1=fopen("TEXTFILE","r");
 file2=fopen("TEXTFILY","w+");

 while((c=fgetc(file1))!='\0')  /* last character in file is '\0' */
  {
    if(c!='\n') /* to bypass new line character automatically
                   inserted by C/400 when reading a line     */
    {
     fputc(c,file2);
     j=j+1;
     if(j==10)
      {
       fputc('\n',file2);  /* write '\n' for every 10 characters */
       j=0;
      }
    }
  }

   fputc('\0',file2);

   fclose(file1);
   fclose(file2);
}
```

Specifying Record Length in a Text File

To specify record length for a text file, the fopen statement should be coded as:

```
file1 = fopen("TEXTFILE","w+ lrecl = ??");
```

where ?? is the length of record.

Text File and Binary File are the Same from AS/400 Viewpoint

From the AS/400 viewpoint, a text file and a binary file are the same. Both are physical or logical files on AS/400. They are different only when files are opened in C/400 programs in different formats. Therefore, a text file when opened as a binary file can be processed like any C/400 binary file. A binary file, when opened as a text file, can be processed like any text file.

Major Differences in Processing Text File and Binary File

In a text file, no record buffer needs to be defined in the C/400 program, as usually the reason for using a text file is because the user wants to process data character by character. The fgetc() function will pass character by character to the program. In writing to a text file, data always starts at the 13th byte of the record. If data is a line read from the screen, a record will be written to the text file when the Enter key is pressed and the maximum number of characters that can be entered each time is 160. Therefore, if a text file is opened for capturing data entered from a screen and taking the default record length, only 160 bytes out of 266 bytes in a record can be used. When reading data from a text file, C/400 will ignore all the preceding and trailing spaces and a new line character '\n' will automatically be added to the end of the input data. A text file can be dynamically created.

When processing a binary file, a record buffer must be defined in the C/400 program (usually in form of a structure). I/O is performed on a record basis. If the binary file is an output file, it must exist before the fopen statement is executed. All leading and trailing spaces in the record will be retained and no new line character '\n' will be added at end of input data.

4.4 Accessing an Externally Described File in a C/400 Program

C/400 does not provide the option to copy a field definition from an AS/400 externally described file into a program. A separate record buffer area must be defined to serve a read/write operation. It is important to be aware that C/400 handles data differently as compared with AS/400 data management and some additional work will have to be done in reading writing records from an AS/400 database file.

Character String

In C/400, a string is represented by an array and is defined in the form of s[n] where n is the length of the character string. The first element of s contains an address pointing to a space in the memory, and that space in memory contains the real content of the string that will be used in the application. Each character in a character string can be referred by specifying the element in the string. For example, s[0] refers to the first element (or first character) in the character string and s[1] refers to the second element (or second character) of the character string.

All C/400 string are terminated by NULL character (HEX 00). When C/400 reads a string, it ends when a NULL character is detected. For example, if string s is pointing to a memory space that contains the following data:

ABCDEFGH12345\0 where \0 is the NULL character

```
printf("%s",s);
```

will display "ABCDEFGH12345".

If the data is "ABCDEFGH\012345\0"

```
printf("%s",s);
```

will display "ABCDEFGH" since '\0' is detected after H.

Similarly, if t is a string of 10 characters and

```
scanf("%s",t);
```

is used to read the string from screen. If the operator entered "AAAABBBB" then pressed Enter, the contents of it will be "AAAABBBB\0".

By now, we can anticipate that records read from AS/400 database into C/400 program may not be used directly. Records written by C/400 programs to an AS/400 database file may have different format as compare with a record written by other languages such as COBOL or RPG. For example :

Sample AS/400 database file and C/400 structure

AS/400 Database File		C/400 Structure
R FORMAT1		struct frecord {
CNUM	5	char cnum[5];
CNAME	30	char cname[30];
CADD	30	char cadd[30];
CTEL	10	char ctel[10];
		} frecord;

If a C/400 program using the structure frecord to read the database file, and printf() statement is used to display the contents of cnum, then the contents of cnum, together with whatever data immediately follows will be displayed until a NULL character (HEX 00) is encountered. If ctel in frecord contains the same information as t (that is it contains "AAAABBBB\0") and is written to a database file, when a COBOL or RPG program attempts to read this record and display it on a display file, a data error will occur since a 5250 terminal cannot display a NULL character.

As a reminder to the developer, when attempting to manipulate a field read from AS/400 database in a C/400 program, make sure the string is terminated with NULL character and this can be done by explicitly assigning a NULL

character to the last element in the string. When writing records to an AS/400 database file, the developer should ensure that all NULL characters have been replaced by spaces.

Numeric Data

In AS/400, the most common numeric data types used are packed decimal and zoned decimal. When defining database file, the user can specify the length of the numeric field.

In C/400, length of numeric data cannot be specified. There are many numeric data type the user can define, but there are no data type such as packed or zoned decimal.

Since the numeric data type of AS/400 database and C/400 are not compatible (except floating point numbers), numeric fields read from AS/400 database cannot be processed directly in C/400 programs and numeric fields generated by C/400 programs cannot be written to database file directly. Some data conversion routine must be used to do the data conversion. Such functions are available as library functions grouped under the EPM Application Library.

The immediate result of this is that numeric data will be converted, formatted and then passed between AS/400 database file and C/400 program as alphanumeric files. As an example, a C/400 program will define such a structure to read/write record from an AS/400 database:

Example of AS/400 Database and Corresponding C/400 Structure :

```
        AS/400 Database                C/400 Structure
------------------------------  ------------------------------
    R FORMAT1                       struct frecord {
      CNUM         5                    char cnum[5];
      CNAME       30                    char cname[30];
      CAMT         8 0                  char camt[5];
                                       } frecord;
```

When the database record is read into the record buffer frecord, content stored in frecord.camt must go through a conversion routine QXXPTOI to convert camt into integer for calculation.

Further detailed information on the C/400 Data conversion routines can be found in this book Section 3 "EPM Application Library" or in this section "Alignment of data" or Appendix B of the *C/400 Users Guide*.

C/400 Program and Record Format of Externally D6scribed Fil

If the C/400 program works on an AS/400 externally described file with only one record format (for example a physical file), there is no concern about which format should be used to read or write records. However, when working on printer files or display files where several record formats can be selected, the C/400 program must explicitly specify which format to use in each read/write. This is accomplished by using the QXXFORMAT statement in the program.

Further information on QXXFORMAT can be founded in Appendix B of *C/400 Users Guide*. Examples on how to access printer file, display file and database file with QXXFORMAT can be found in the Sample Programs in the Appendix of this manual.

C/400 Files or AS/400 Externally Described File?

The reader probably will agree by now that additional work is required if a C/400 program is to interface with AS/400 database files. The same will be true when a program written in COBOL or RPG needs to access a file with records generated by C/400 program. Data conversion and formatting is almost a "must" in such a situation. But if a file will only be used by C/400 programs, then there is no need to do all the data conversion discussed here. Just let the C/400 program handle the data in its own convention and development effort is reduced.

4.5 How to Use AS/400 Database Files in C

This chapter discusses how to describe AS/400 database files in C/400 language and how to manipulate data in existing database files.

Database Files on the AS/400 can be described at a record level or on a field level. This chapter discusses only how to use files described at a field level using DDS (data definition specifications) or using SQL/400. SQL is the SAA CPI database interface implementation for the AS/400.

Database files created with a field level description are referred to as externally described files. The main advantage of an externally described file is that the field descriptions are not stored in the HLL program. The AS/400 data management separates the data description from the program. However, in C/400 you must describe a structure to define how the record will appear. C/400 does not provide a function to reference to external data descriptions.

There are different methods to describe an external file.

- Data Definition Specifications (DDS)

  Can be used for the definition of physical files (PF), logical files (LF), display files, printer files and ICF files. Use the Source Entry Utility (SEU) to create the DDS source code. To create the file object issue a CRTxxx CL command and specify the name of the new file and the name of the DDS source member.

- Interactive Data Definition Specifications (IDDU)

  Is primarily designed for compatibility with IDDU/36 so that AS/400 users can easily port their applications from a System/36 to the S/36 environment and that these users have available the same facility as they may have used before. IDDU is an interactive application which prompts the user to define the characteristics of data files and fields on AS/400.

- Data Definition Language statements of SQL/400.

  There are two basic types of SQL statements: data definition statement (DDL) and data manipulation statements (DML). The following statements are part of the SQL DDL:

  - CREATE COLLECTION (former CREATE DATABASE)

- CREATE TABLE
- CREATE VIEW
- CREATE INDEX.

SQL DDL statements can only operate on objects created by SQL in an SQL collection (database). However you can mix DDS and SQL DDL. Refer to *SQL/400: A Guide to Implementation* and to *SQL/400 Programmer's Guide.*

The following two examples show a field level record definition. The first is an example of the Display File Field Description (DSPFFD) Command for file MATABLE described with DDS. The field attributes for this file are defined in a field reference file. The second is an example of the SQL Statement CREATE TABLE to create the same file with SQL. SQL/400 does not support the data type NULL. For this reason we have specified 'NOT NULL WITH DEFAULT'.

```
  File . . . . . :  MATABLE
    Library . . . . . :  SQLDEMO
File Information
 :
Record Format Information
 :
Field Level Information
            Data      Field  Buffer    Buffer       Field
  Field     Type     Length  Length  Position       Usage    Column Heading
  KZDST     CHAR          1       1         1       Both     Dienstellen-
                                                             Kennzeichen
    Field text . . . . . :  Dienstellen- Kennzeichen
    Referenced information
      Referenced file . . . . . :  REFTABLE
        Library . . . . . :  DBDEMO
      Referenced record format . . . . . :  REFTABF
      Referenced field . . . . . :  KZDST
      Attributes changed . . . . . :  None
            Data      Field  Buffer    Buffer       Field
  Field     Type     Length  Length  Position       Usage    Column Heading
  EART      CHAR          1       1         2       Both     Empfinger-
                                                             Art
    Field text . . . . . :  Empfinger- Art
            Data      Field  Buffer    Buffer       Field
  Field     Type     Length  Length  Position       Usage    Column Heading
  HK        CHAR          2       2         3       Both     Hauptkasse
    Field text . . . . . :  Hauptkasse
            Data      Field  Buffer    Buffer       Field
  ENR       CHAR          6       6         5       Both     Empfanger-
                                                             Nummer
    Field text . . . . . :  Empfanger- Nummer
    Referenced information
            Data      Field  Buffer    Buffer       Field
  Field     Type     Length  Length  Position       Usage    Column Heading
  NAME      CHAR         40      40        11       Both     Name
    Field text . . . . . :  Name
            Data      Field  Buffer    Buffer       Field
  Field     Type     Length  Length  Position       Usage    Column Heading
  VNAME     CHAR         20      20        51       Both     Vorname
    Field text . . . . . :  Vorname
            Data      Field  Buffer    Buffer       Field
  Field     Type     Length  Length  Position       Usage    Column Heading
  GEBDAT    ZONED      8  0       8        71       Both     Geburts-
    Field text . . . . . :  Geburts- Datum
            Data      Field  Buffer    Buffer       Field
  Field     Type     Length  Length  Position       Usage    Column Heading
  KIZAHL    ZONED      2  0       2        79       Both     Kinderzahl
    Field text . . . . . :  Kinderzahl
 :
```

```
Table Creation of MATABLE in SQL
   CREATE TABLE MATABLE
               ( KZDST  CHAR(1)   NOT NULL WITH DEFAULT,
                 EART   CHAR(1)   NOT NULL WITH DEFAULT,
                 HK     CHAR(2)   NOT NULL WITH DEFAULT,
                 ENR    CHAR(6)   NOT NULL WITH DEFAULT,
                 NAME   CHAR(40)  NOT NULL WITH DEFAULT,
                 VNAME  CHAR(20)  NOT NULL WITH DEFAULT,
                 GEBDAT DEC (8,2) NOT NULL WITH DEFAULT,
                 KIZAHL DEC (2,0) NOT NULL WITH DEFAULT,
                 STAATA CHAR(3)   NOT NULL WITH DEFAULT,
                 PLZ    CHAR(8)   NOT NULL WITH DEFAULT,
                 ORT    CHAR(30)  NOT NULL WITH DEFAULT,
                 STR    CHAR(30)  NOT NULL WITH DEFAULT,
                 TEL    CHAR(15)  NOT NULL WITH DEFAULT,
                 EBERUf CHAR(40)  NOT NULL WITH DEFAULT
               )
```

The next example shows the definition of a C structure for the database file defined above. Note that:

- Strings in C/400 are terminated with the null character \0 . Field (column) 'name' has type character with a length of 30. The corresponding C/400 variable 'name' is defined as character with a length of 31.
- C/400 does not support decimal numbers. Column (Field) 'kizahl' is defined as
  - ZONED 2,0 in DDS
  - DEC (2,0) in SQL
  - DOUBLE in C/400

```
Record Format Description in C/400
   struct marec { char   kzdst   [2];
                  char   eart    [2];
                  char   hk      [3];
                  char   enr     [7];
                  char   name    [41];
                  char   vname   [21];
                  double gebdat;
                  double kizahl;
                  char   staata  [4];
                  char   plz     [9];
                  char   ort     [31];
                  char   str     [31];
                  char   tel     [16];
                  char   eberuf  [41];
          } marec;
```

Reading a Record from an Externally Described File

Before you can read records from a database file you have to open the file using the 'fopen' or 'freopen' function with an valid open mode. Open mode 'rb' performs the data management operation 'Open for Input'. For processing a record at a time use the keyword 'type' with parameter 'record'. Opening the file with mode 'rb' causes a lock condition shared for read (*SHRRD) on the data.

The QXXFORMAT routine is a AS/400 extension to the C language. Using files where more than one record format can be selected, such as logical, printer or display files, it is necessary to set the record format name by using QXXFORMAT. The record format name must have a length of 10 characters.

Use the 'fread' function to read records sequentially. If the file is described as a keyed file you read in the keyed order; otherwise read in arrival sequence.

The functions fopen, freopen, fread are provided in the standard input/output include file <stdio.h>. Refer to the *C/400 User's Guide* for valid open modes and more information.

```
main()
{

  :
   MATABLE = fopen("SQLDEMO/MATABLE", "rb type=record");
   QXXFORMAT(MATABLE,"MATABF    ");
   fread(&satz,sizeof(satz),1,MATABLE);

  :
   fclose(MATABLE);
}
```

Writing a Record to an Externally Described File

The open mode 'ab+' specified in the 'fopen' function performs an 'Open for Input and Output' for the specified file. Opening the database file with mode 'ab+' causes a lock condition shared for update (*SHRUPD) on the data. The QXXFORMAT routine is an AS/400 extension to the C language. Using files where more than one record format can be selected, such as logical, printer or display files, makes it necessary to set the record format name by using QXXFORMAT. The record format name must have a length of 10 characters.

The 'fwrite' function appends the record at the end of the file. Refer to the *C/400 User's Guide* for more information.

```
main()
{
  :
   struct satz satz;
   MATABLE = fopen("SQLDEMO/MATABLE", "ab + type=record");
   QXXFORMAT(MATABLE,"MATABF    ");
  :
   fwrite(&satz,sizeof(satz),1,MATABLE);
   fclose(MATABLE);
 }
```

Using Existing AS/400 Data

The matrix shown in the following figure must be considered carefully when using a C program with existing externally described data.

Table 1. DDS-to-C/400 Data Type Mapping

DDS Data Type	Length	Decimal Position	Buffer Length	C/400 Declare
Indicator	1	0	1	char
A	1	none	1	char xxx (where xxx is field name)
A	2-32766	none	2-32766	char xxx [n] (where n=2 to 32766)
B	1-4	0	2	short int
B	1-4	0	2	unsigned int xxx:16 (need alignment)
B	5-9	0	4	int
B	5-9	0	4	unsigned int xxx:32 (need alignment at a word boundary)
B	1-4	1-4	2	char xxx[2]
B	5-9	1-9	4	char xxx[4]
P	1-31	0-31	1-16	char xxx[n] (where n = length/2 + 1)
S	1-31	0-31	1-31	char xxx[n] (where n = 1 to 31)
F	1-7	0-7	4	float (need alignment at a word boundary)
F	8-15	0-15	8	double (need alignment at a word boundary)

Character Data

In the C language a string is not a separate data type; instead it is an array of the type 'char'. The last character of a string is the character '\0'. This special character, the 'null character', occupies one byte of memory and has the value hex(00). (The null character in C has a different meaning as the data type NULL in SQL.) A character array not terminated with a "null character" is not recognized as a string. It is the only way string functions provided in the include file <string.h> can know where the end of the string is.

Whenever you read data from an existing file and you want to use string functions provided by C, make sure that the character array is terminated by the "null character". As the null character is included in the length, specify a string whose length is n + 1.

To avoid the problem with the terminating \0 you can use the memxxx functions instead of string functions (for example, memcpy).

For example:
Field NAME in file MATABLE has data type CHAR and field length 40.
The corresponding C variable is defined as a name with a length of 41.

Numeric Data

Zoned and packed data types are not supported in C/400. Use the data conversion routines provided in the EPM Application Library. You can use logical files for an override from packed and decimal data to floating point. Refer to the following section for this method. C/400 expects that floating point numbers are aligned to a word boundary.

Usage of Logical Files for Data Type Overrides

You can override the data type specified in the physical file definition using a logical file. For every logical record format you must specify a record format name and the PFILE or JFILE keyword. The file names specified on these keywords are the physical files that the logical file is based on. When a record is read from the logical file, the fields from the physical file are changed to match the logical file description. If the program updates or adds a record, the fields are changed back. The following table shows the possible conversions between physical and logical files. You can use the logical file also to override the data type of an table created with the SQL statement CREATE TABLE, but the object created by the CRTLF command can not exist in an SQL collection. That means the logical file must exist in an non-SQL library.

Table 2. Data Conversion between Physical and Logical Files					
	Logical File Data Type				
Physical File Data Type	Character	Zoned	Packed	Binary	Floating Point
Character	Valid	Note 1	Not valid	Not valid	Not valid
Zoned	Note 1	Valid	Valid	Note 2	Valid
Packed	Not valid	Valid	Valid	Note 2	Valid
Binary	Not valid	Note 2	Note 2	Note 3	Note 2
Floating point	Not valid	Valid	Valid	Note 2	Valid

Note:

1. Valid only if number of characters equal to number of digits
2. Valid only if the binary field has a decimal precision of zero
3. Valid only if both fields have the same decimal precision

Mapping between floating point fields and other numeric fields may result in rounding or a loss of precision.

Refer to *Data Definition Specifications Reference* for more information on data type conversion.

4.6 Open Query File Usage

The AS/400 Command Language provides a command that allows you to process data base functions similar to DDS and the Create File CL commands (CRTPF, CRTLF). The Open Query File command (OPNQRYF) enables a subset of records from a file to be selected for use during program execution. OPNQRYF must be used in a HLL program.

Functions of OPNQRYF are:

- Record selection
- Grouping and sorting
- Calculations
- Dynamic join
- and more.

Refer to the *AS/400 CL Reference Vol. 1 - 5* for a complete list of functions.

OPNQRYF is useful as a programmer's tool to improve efficiency of programs. It has more functions to select data and to calculate with the data than AS/400 Query and SQL/400. However, it is not part of System Application Architecture (SAA) and if your goal is to write a portable application you should not use the OPNQRYF command.

The following example shows how to code the OPNQRYF command using the C/400 interface to the AS/400 command processor:

```
main()
{

  :
 /* Override Data Base File with share Open Data Path = YES */
 system("OVRDBF MATABLE SHARE(*YES)");
 system("OPNQRYF FILE(MATABLE)");

   MATABLE = fopen("SQLDEMO/MATABLE", "rb type=record");
   fread(&satz,sizeof(satz),1,MATABLE);
   while (!feof(MATABLE))
   {

  :
     fread(&satz,sizeof(satz),1,MATABLE);
   }
 /* Close Opend File MATABLE                    */
 system("CLOF OPNID(MATABLE)");
 /* Delete Override                          */
 system("DLTOVR FILE(MATABLE)");

  :
}
```

Refer to Chapter 9 in the *AS/400 Data Base Guide* for an intensive discussion of the OPNQRYF command.

4.7 Commitment Control

The AS/400 has an integrated transaction recovery function called commitment control. This means that you can group database operations as a single transaction. You can ensure that complex data base operations, if the job ends abnormally, are synchronized and the database integrity is correct. There are three levels of commitment control. These are *NONE, *CHG, *ALL. For more information on journal management and transaction recovery see the *AS/400 Backup and Recovery Guide*.

There are three different ways of using commitment control on the AS/400 with the C language. After setting up your commitment control environment use the Start Commitment Control command (STRCMTCTL) before calling your C application.

For the C/400 Commitment Control Functions there are two routines provided in the EPM Application Library: QXXCOMMIT and QXXROLLBCK. The file must be opened with the 'commit=y' keyword parameter on the fopen function. Refer to Chapter 5 in the *C/400 User's Guide* for how to use these C/400 extensions in your program.

The OS/400 provides the two commands COMMIT and ROLLBACK. You can use these CL commands through the C system function.

```
main ()
{

 :
  system("COMMIT CMTID(*NONE)");

 :
  system("ROLLBACK");
}
```

The third way to use commitment control is SAA-conforming. If COMMIT(*CHG) or COMMIT(*ALL) is specified when the program is compiled, SQL automatically sets up the commitment control environment by implicitly invoking the Start Commitment Control command. You can use the functions provided by SQL/400 as in Example 10 in Appendix B.

```
main ()
{

 :
  EXEC SQL COMMIT;

 :
  EXEC SQL ROLLBACK;
}
```

4.8 Coding SQL - Considerations of Usage

The System Application Architecture Common Programming Interface (SAA CPI) defines the SQL/400 with the OS/400 relational data base as the interface to define and access. The usage of SQL/400 is essential when it is required to write a portable application. Since C/400 provides no keyed access to the AS/400 database SQL is one way to retrieve data from the database.

Before executing a C program with embedded SQL statements there are two steps to create the program object:

- Use the CRTSQLC command to create a C/SQL program. This command calls the SQL pre-compiler to compile the SQL statements embedded in your C program. If you are using the Programming Development Manager (PDM) the source member type is SQLC. This type is recognized and provides several functions by PDM such as the correct choice of compiler.

```
                    Create SQL C Program (CRTSQLC)
Type choices, press Enter.
Program . . . . . > EXAMPLE       Name
  Library . . . . . .   *CURLIB     Name, *CURLIB
Source file . . . . . .  QCSRC      Name, QCSRC
  Library . . . . . .   *LIBL      Name, *LIBL, *CURLIB
Source member . . . . . .  *PGM     Name, *PGM
Commitment control . . . . . .  *CHG  *CHG, *ALL, *NONE
Text 'description' . . . . . .  *SRCMBRTXT
                    Additional Parameters
Precompiler options . . . . . .  *NOSRC  *SRC, *SOURCE, *NOSRC...
          + for more values
INCLUDE file . . . . . .  *SRCFILE  Name, *SRCFILE
  Library . . . . . .   *LIBL      Name, *LIBL, *CURLIB
Severity level . . . . . .  10      0-40
Source margins:
  Left margin . . . . . .  *SRCFILE  1-80, *SRCFILE
  Right margin . . . . . .          1-80
Print file . . . . . .  QSYSPRT     Name
  Library . . . . . .   *LIBL      Name, *LIBL, *CURLIB
```

  Refer to the *SQL Programmer's Guide* for further explanations of the command parameters.

- After a successful pre-compile a temporary source file member with the same name as the program is created in source file QSQLTEMP in QTEMP. By default the C compiler is called after the pre-compile process to compile the temporary source member. Refer to Chapter 3 for information on how to use the debug for a C/SQL program.

C Host Variables for SQL

Host variables are necessary to receive the retrieved data from a SELECT statement or to specify a condition in the WHERE clause. C/400 and SQL/400 allow the user to declare host variables that are pointers. Pointers are a mechanism to access a data object without referring to the data object (or function) directly. The pointer holds the address of a variable or a function. For restrictions refer to the *SQL Programmer's Guide* Chapter 8.

The following example is a C structure used as a host variable. The description matches the record format of the SQL table referred to earlier. The structure marec is specified in the INTO clause of the FETCH statement in the program example for static SQL on the next pages.

```
struct marec {
        char    kzdst   [2];
        char    eart    [2];
        char    hk      [3];
        char    enr     [7];
        char    name    [41];
        char    vname   [21];
        double  gebdat;
        double  kizahl;
        char    staata  [4];
        char    plz     [9];
        char    ort     [31];
        char    str     [31];
        char    tel     [16];
        char    eberuf  [41];
    } marec;
```

The following SQL statement shows how you code host variables in an embedded SELECT statement. The preceding colon declares that structure marec is a host variable. The unique index 'enr' specifies the condition in the WHERE clause.

```
EXEC SQL
SELECT *                /* retrieve all columns */
INTO :marec
FROM MATABLE
WHERE ENR = :marec.enr;     /*unique index       */
```

4.9 Using Existing Data in SQL/400 Tables

Table 3 (Page 1 of 2). Data Types SQL and C/400

SQL Data Type	Buffer Length	C Equivalent	Comment
CHAR (1)	1	char identifier	A single character.
No equivalent (char string length n)	n	char identifier [m]	char array to hold NUL terminated strings (m=n+1)
Varying-length char string		struct { short len; char s[n] } identifier;	structure to emulate varying-length string.
SMALLINT	2	short int	16 bit, signed integer
INTEGER	4	long int	32 bit, signed integer

Table 3 (Page 2 of 2). Data Types SQL and C/400			
SQL Data Type	**Buffer Length**	**C Equivalent**	**Comment**
DECIMAL	1-16	No equivalent	C does not support decimal numbers. Code a decimal column as float, double or integer. See comment.
REAL (single precision floating point)	4	float identifier	floating point (needs alignment)
FLOAT (double precision floating point)	8	double identifier	floating point (needs alignment)
NUMERIC (zoned decimal)	1-31	No equivalent	C does not support zoned numbers. Code a numeric column as float, double or integer. See comment.

- FLOATING POINT NUMBERS

  There is a difference between the floating point formats used by C/400 and SQL/400. Floating point numbers in C/400 need alignment. This fact is usually not required in a data base. Use views (or logical files) where the floating point numbers are aligned to a word (4 byte) boundary in the record buffer.

- PACKED DECIMAL and ZONED DATA TYPE

  As zoned and packed decimal formats are not supported in this version of C/400, use the EPM Application Library Conversion Routines as described in Appendix B of the *C/400 User's Guide*. The following conversion routines are provided:

  - QXXDTOP floating point to packed decimal
  - QXXDTOZ double to zoned decimal
  - QXXITOP integer to packed decimal
  - QXXITOZ integer to zoned decimal
  - QXXPTOI packed decimal to integer
  - QXXPTOD packed decimal to double
  - QXXZTOD zoned decimal to double
  - QXXDTOI zoned decimal to integer.

- CHAR TYPE

  A string in C is not a separate data type. Instead it is an array of type char terminated by the null character (\0). The null character is included in the length. Whenever you read data from an existing table and you want to use string functions provided by C, make sure that the character array is terminated by the null character and add one byte to the length of the host variable. Instead of string functions memxxx functions can be used (for example, memcpy) to avoid the problem with the terminating '\0' character.

  For example if an SQL column has a data type char and field length 40, define the corresponding host variable with a length of 41.

The user may also use the override possibility of a DDS logical file as described earlier. However note that a logical file cannot exist in an SQL collection (database). The logical file for data type conversion must be in a non-SQL library. However it is possible to create a logical file based on a SQL table. See section 'Using Logical Files for Data Type Overrides' in this document.

Static SQL in a C/400 Program

There are two different ways to process static SQL statements in an C program: Either with a CURSOR or not.

If the result of the processed statement is a single row you don't have to declare a CURSOR. (for example, if your SQL table is accessed by a unique index).

If the result table of a SELECT statement can contain multiple rows matching the select criteria of the WHERE clause, then a CURSOR must be used to make single rows of the result table available to your program. The C language, as well as other HLLs, is not able to process a set of rows (records) at a time.

The SELECT statement in the following figure retrieves all columns from table MATABLE into the CURSOR C1. The C variable "matchcode" specifies the value for retrieving rows in the LIKE clause of the SELECT statement. The result table contains the rows which match the select criteria and the cursor points to the current row (in this example the first row). The FETCH statements retrieves the rows sequentially into the C structure "marec" until end-of-data.

Note that the "EXEC SQL" expression must be all on one line. The rest of the SQL statement may be on more than one line.

```
     char  matchcode  8 ;

  :
     EXEC SQL INCLUDE SQLCA;
main () {
  :
     EXEC SQL  WHENEVER SQLERROR GO TO error_cursor;
     EXEC SQL  DECLARE C1 CURSOR FOR
             SELECT *
             FROM MATABLE
             WHERE NAME LIKE  :matchcode
     EXEC SQL  WHENEVER SQLERROR GO TO error_open;
     EXEC SQL  OPEN    C1;

   while (SQLCODE == 0) {
      EXEC SQL  WHENEVER SQLERROR GO TO error_fetch;
      EXEC SQL
      FETCH C1 INTO :marec;
  :
     }
  goto ende;
 error_cursor:
  printf("Error at  DECLARE CURSOR %d??/n",sqlca.sqlcode);
  goto ende;
 error_open:
  printf("Error at  OPEN    CURSOR %d??/n",sqlca.sqlcode);
  goto ende;
 error_fetch:
  printf("Error at FETCH %d??/n",sqlca.sqlcode);
 ende:
   EXEC SQL  CLOSE   C1;
```

Using Dynamic SQL in C/400

This section discusses the sample C program using the dynamic SQL programming technique as shown on the next pages and in Appendix B. For additional information on this technique the reader should refer to AS/400 publications and newsletters.

Other than using SQL in a static environment, in dynamic SQL the embedded statements are prepared and executed at program run time. At compile time the SQL pre-compiler does not need to know what kind of SQL statement will run while executing the program. When using static SQL the SQL optimizer creates an access plan during precompile time that tells the system how to run the embedded SQL statements in the most effective way. In dynamic SQL the optimizer cannot build this access plan, because references to tables and views do not exist. The access plan has to be built at run time. This affects the performance of the application.

We differentiate between two basic types of embedded SQL statements: SELECT and non SELECT statements. The data that a non SELECT statement returns to the program is a return code in the SQL communication area. The

example program uses non SELECT statements, DELETE and UPDATE, in a dynamic programming technique. It allows you to run these two statements similar to an interactive SQL session. The functions are restricted to the tables, columns and operators shown in the display file below.

Use the DSPFFD command to look at the input and output buffer of your display file. You have to code a structure in your C program that matches the input and output records. If you are using indicators you have to decide if they exist in the input and output records or in a separate indicator area (INDARA keyword in your DDS definitions).

The following figure shows one of the screens used in the example program:

```
                Dynamic SQL in an C/400 Program

                      Prepare  -  Mode

DELETE  FROM _________                    WHERE _______    __          ?
             MATABLE                            VNAME      =
                                                NAME       >
                                                STAATA     <
                                                PLZ        >=
                                                ORT        <=
                                                STR
             KITABLE                            KNAME
                                                KKIGEL


   Status of the prepared SQL statement
   DELETE FROM 12345678                 WHERE 123456 12 ?

Function keys:  3=Exit  8=Prepare Update  9=Prepare Delete  10=Execute
                11=Rollback  12=Commit
```

The display file allows you to specify tables, columns and operators to prepare the SQL statements UPDATE or DELETE. You can switch between the statements via function keys. Pressing function keys you can explicitly process COMMIT or ROLLBACK. This is not the usual way commitment control is designed in an application but shows the coding required for these functions. The line under "Status ..." shows how the the statement looks in the C variable. The "?" is the parameter marker for the actual value for executing the statement using Function key 10. The digits (123..) are replaced by the values you type in for table and column names.

Following is the pointer definition for the externally described display file. Processing of display files is discussed in Chapter 5. For the definition of the input and output buffers see the DDS source for the display file and the complete program in Appendix B.

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

    /*****************************/
    /* Global Data declarations */
    /*****************************/

    FILE *dspf1;          /* Pointer to locate display file*/
  :
```

Every C/400 program using embedded SQL statements must have an SQL communication area (SQLCA). The SQLCA must be embedded before any SQL or C statement can be run. In this example the INCLUDE SQLCA statement is used. For information how to code a C variable refer to Chapter 8 of the *SQL Programmer's Guide*. The SQLCA provides information for error handling in your program.

```
  :
    EXEC SQL INCLUDE SQLCA;
  :
```

The following C structures are used to build that SQL statement that are required. In the program sample SQL statements are used which contain parameter markers indicated by a "?". The user may set up the PREPARE section first and then run it using the EXECUTE statement with different values every time the statement is executed. Statements with no parameter markers can be run by the EXECUTE IMMEDIATE statement.

```
/** This is the structure for preparing the UPDATE statement **/
   struct xupd
   {
   char upd [7];
   char tbl [8];
   char set [7];
   char col1 [6];
   char eql  [8];
   char usg1 [2];
   char wher [6];
   char col2 [6];
   char opc  [2];
   char usg2 [2];
   };
/** This is the structure for preparing the DELETE statement **/
   struct xdel
   {
   char del [12];
   char tbl [8];
   char wher [24];
   char col1 [6];
   char opc  [2];
   char usg1 [2];
   };
```

C/400 host variables cannot be elements of vectors,structures or arrays except for character arrays representing a string. So you must use a string variable as the parameter of the PREPARE statement. That is the reason why the structures are redefined. The string variable "updat" ("delet") is the update (delete) statement that copied into "obild.s" the PREPARE statement is naming STMT. The element "obild.s" is used in the display file, so that you can follow interactively the steps of the PREPARE process.

```
/* redefine the structure for update as string used by the
   SQL PREPARE statement                                  */
   union uupd {
   struct xupd supd;
   char updstr [56];} updat;

/* redefine the structure for delete as string used by the
   SQL PREPARE statement                                  */
   union udel {
   struct xdel sdel;
   char delstr [56];} delet;
```

```
The program flow of the sample program is:
     Open the display file SQL2D. See Appendix B example 10.
     Initialize indicators
     While not function key 3 do

      write display file
      read display file

      if function key 8   prepupd();
      if function key 9   prepdel();
      if function key 10  EXEC SQL PREPARE;
         if SQLCODE == 0     execstmt();
      if function key 10  EXEC SQL ROLLBACK;
      if function key 10  EXEC SQL COMMIT;

   End of while loop
```

```
main()
{
  :
/* carry on if F3 is not hit by user                    */
   while ( ibild.in03 != 0XF1)
     {
/* Display sql2d                                        */
   QXXFORMAT(dspf1,"FMT1      ");
   fwrite(&obild,sizeof(obild),1,dspf1);
   fread(&ibild,sizeof(ibild),1,dspf1);
  :
          if (ibild.in10 == 0XF1 )
            {
             EXEC SQL
             PREPARE STMT FROM :obild.s;   /* PREPARE */
             if (sqlca.sqlcode == 0)
               {
                obild.in63 = 0XF1;
               } /* sqlcode == 0 */
             else/* SQL Error ?  */
               {
                obild.in88 = 0XF1;
                obild.sqldec = sqlca.sqlcode;
               }                    /* end SQL Error */
            } /* in10 == 1 */
          else
            {
             if (ibild.in11  == 0XF1)
               {
                EXEC SQL
                ROLLBACK;              /* ROLLBACK     */
                obild.in63  == 0XF0;
               }                     /* in11 == 1     */
             else                     /* in11 != 1     */
               {
                if (ibild.in12 == 0XF1)
                  {
                   EXEC SQL
                   COMMIT;             /* COMMIT       */
                   obild.in63  == 0XF0;
                  }                  /* in12 == 1     */
                 else {              /* in12 != 1     */
                   if (obild.in63 == 0XF1)
                         execstmt();
                   else {            /* in63 != 1     */
                     if (obild.in61 == 0XF1)
                           prepupd();
                     else            /* in61 != 1     */
                     {
                       if (obild.in62 == 0XF1)
                             prepdel();
  :
}                                    /*     main     */
```

As the statement is prepared completely the function "execstmt" is called to execute the UPDATE or the DELETE. The parameter markers will be replaced by the host variables "ibild.frm" and "ibild.whe". If you want to update with different values in the SET clause the statement has to be prepared only once.

If you want to switch between the UPDATE and the DELETE function, the statement has to be prepared again.

```
/******* Execute Statement  *********************************/
void execstmt()
{
  if (obild.in61 == 0XF1)
    {
    EXEC SQL
    EXECUTE STMT USING :ibild.frm, :ibild.whe;
              if (sqlca.sqlcode != 0)
                {
  :
                } /* sqlcode != 0 */
    }
  else
    {
     EXEC SQL
     EXECUTE STMT USING :ibild.whe;
     if (sqlca.sqlcode != 0)
       {
  :
       } /* sqlcode != 0 */
    }
 }  /* execute statement function */
```

C/400 and SQL/400 are SAA-standard products. SQL/400 does not yet completely match the SAA database reference definitions, but it will be consistently implemented across the SAA environments. C/400 supports the SAA C-Level 2 and the ANSI standard with AS/400 extensions. Portable applications among other SAA environments can be one reason to use embedded SQL. Another reason may be the incompatibility of data types between C language and existing data on the AS/400.

5. Display File Processing

In C/400 the default input/output device for the program is the display terminal. Functions like printf(), scanf(), getchar() accept input or display output on the terminal in a line mode and the device need not be "opened" for such operation. When the application requires full-screen processing, display files are required.

Display files under C/400 are handled similar to disk files. An fopen statement is required to associate the display file to the program. Statements on fread(), fwrite() will be used to read from display files and write to display files. If the display file contains multiple record format, QXXFORMAT will be used to specify which format the fread and fwrite will use. C/400 provides support for message subfile but there is no support on other functions of subfile.

It is important to remember that display files and indicators apply to C/400 on AS/400 only. C programs developed using display files and indicators are not portable to other systems.

5.1 Usage of Indicators

There are two ways to use indicators in a display file. Indicators can be handled in a different area by using INDARA keyword. Indicators can also be treated as part of data in the input and output buffer. Indicators can have only two values - '0' or '1'. In C/400, indicators can be set to value 0XF0 or 0XF1 or simply '0' or '1'. For example:

```
Setting indicators in C/400

main()
 {
  char in01;
  char in02;
  char in03;
  char in04;

    in01 = 0XF0 ; /* set indicator in01 to '0' */
    in02 = 0XF1 ; /* set indicator in02 to '1' */
    in03 = '0' ;  /* set indicator in03 to '0' */
    in04 = '1' ;  /* set indicator in04 to '1' */
 }
```

In C/400, indicators can be defined as a character variable and they can be set to specific values as shown in above example.

Using Indicators with the INDARA Keyword in Display Files

When handling indicators in a separate area with INDARA keyword, the display file automatically sets aside 99 bytes to store information about the 99 indicators. This information will be passed to and from the C/400 program. In the C/400 program, an area of 99 bytes must be declared and associated it with the indicator area of the display file. The following illustrates how to define the DDS for a display file and C/400 program:

Defining INDARA in DDS of Display File

```
                        DSPSIZ(24 80 *DS3)
                        INDARA
                        CA01(01 'End of Job')
                        CA02(02 'Refresh')
                        CA03(03 'Scroll Forward')
                        CA04(04 'Scroll Backward')
                        CA05(05 'Previous')
      R FORMAT1
        CNUMBER   5A B 2  6COLHDG('Customer' 'Number')
          "          "          "
          "          "          "
```

Defining indicators in C/400 program

```
main()
 {
  union     /* meaning to share two or more structure with the
               same memory location                          */
  {
    struct
     {
      char in01;
      char in02;
      char in03;
      char in04;
      char in05;  /* indicator 01 to 05 are defined separately */
      char in06[94];  /* indicator 06 to 99 are defined in an
                         array if the program does not use
                         any of them                        */
       } str;
    char array[99];   /*  array is defined here for east of
                         initialization                     */
     } indic;

  FILE *file1;

  file1 = fopen("DSPFILE","ab+ type=record indicators=y");
                /* indicator = y means display file will use
                   a separate area to store indicator information.
                */

   QXXSINDARA(file1,indic.array);
                /* QXXSINDARA relates the display file and
                   its indicator information to structure
                   indic.array in the C program together */

     indic.str.in01 = 0XF0;
     indic.str.in02 = '0';  /* set indicator 01 and 02 to '1'  */
       "   "
```

Using Indicators as Data in a Display File Input/Output Buffer

Using this approach indicators are ordinary indicator fields in an input/output buffer. The following are examples on how to define indicators using this approach:

DDS of display file with indicators in input/output buffer

```
                                   DSPSIZ(24 80 *DS3)
         R FORMAT1
                                   CA03(03 'End of Job')
           CNUMBER          5A  B 4  6COLHDG('Customer Number')
     99                            ERRMSG('Error : Invalid number')
           "      "           "             "
           "      "           "             "
```

Defining indicators as data in input/output buffer

```
main()
 {
  struct fscrout {
     char in99;   /* variable for indicator 99  */
     char coption;
     } fscrout;

   struct fscrin {
     char in03; /* variable for indicator 03 */
     char coption;
     } fscrin;

FILE *file1;

  file1 = fopen("DSPFILE","ab+ type=record");

   fscrout.in99 = 0XF1 ; /* set indicator 99 to '1' */
       "     "
```

5.2 Use of Conversion Routines

As discussed before, since C/400 does not have numeric data type zoned or packed, any numeric data read from AS/400 database must first be read as character string into the program. The character string will then be converted to one of C/400's numeric data type before the program can use the number for calculation. When a number in a C/400 program needs to be written to a database file or display file, the number must first be converted to character string.

A commonly used conversion routine will be to convert an AS/400 database numeric field to/from C/400 numeric data field. These conversion routines are

available from the C/400 library. Detailed information can be found in Chapter 3 of this manual and Appendix B of the *C/400 User's Guide*.

5.3 Print Key

When the Print key on a terminal is pressed the screen image will be sent to the default printer file for printing.

5.4 Display File Handling in a C/400 Program

Display File and QXXFORMAT

A C/400 program may write various screen formats from a display file to the terminal. QXXFORMAT is used to specify which record format will be used. Example of a QXXFORMAT statement is as follows:

```
QXXFORMAT(file1,"FORMAT1   ");
```

Display File Without Using Indicator

For an example of a C program using display file without an indicator, please refer to Example 6 of the sample programs in the Appendix of this manual.

Display File Using Separate Indicator Area (INDARA)

For examples of using display files with a separate indicator area (INDARA), please refer to Example 8 of the sample programs in the Appendix of this manual.

Display File with Indicator as Data in Input/out Buffer

The following is an example illustrating how the C/400 program works with display files using an indicator as data in an input/output buffer:

DDS for display file

```
                                DSPSIZ(24 80 *DS3)
          R FORMAT1
                                BLINK
                                CA03(03 'End of Job')
                             4 35'Testing Screen'
                                DSPATR(HI)
                                DSPATR(UL)
                            11 10'Enter <E> to End :'
            COPTION     1 B 11 50
     99                         ERRMSG('You have just entered X')
                            15 20'Enter X will show Error Message'
                            17 10'F-3 = End of Job'
```

The following C/400 program writes a screen to the display and accepts input. If an operator enters an "X", indicator 99 will be turned on and an error message

will appear at the bottom of the screen. This function will repeat until the operator enters an "E" or presses Function key 3.

C/400 program working with display file

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
 struct scrout {
    char in99;
    char coption;
     } scrout;

 struct scrin {
    char in03;
    char coption;
     } scrin;

FILE *file1;
int i;
int j;

  file1 = fopen("DSPFILE","ab+ type=record");
  QXXFORMAT(file1,"FORMAT1   ");      /* indicate that FORMAT1 will
                                 be used as the format    */
  clrout(&scrout,sizeof(scrout));     /* fill output buffer with spaces */
    scrout.in99 = 0XF0;               /* set indicator 99 to '0'  */
    while(scrin.coption != 'E')
     {
      fwrite(&scrout,sizeof(scrout),1,file1); /* write screen */
      clrout(&scrout,sizeof(scrout)); /* clear output buffer */
      scrout.in99 = 0XF0;             /* set indicator 99 to '0' */
      fread(&scrin,sizeof(scrin),1,file1);  /* read screen */
      if(scrin.in03 == '1')
        scrin.coption = 'E';          /* F-3 means end of job */

      if(scrin.coption == 'X')
        scrout.in99 = 0XF1;           /* if X is entered, set indicator 99 on */
       scrout.coption = scrin.coption;
   }
  fclose(file1);
}
clrout(buffer,l)                      /* fill output buffer with zero */
char *buffer;
int l;
{
 int i;

  for(i=0;i<l;i++)
   {
    *buffer = ' ';
    *buffer++;
}
```

6. Performance Conclusions and Considerations

6.1 General Observations from Performance Comparison Programs

The following comparisons were derived by using the test programs later in this chapter. For an exact comparison it is suggested that the user could use these programs as a sample. The exact times have not been included because the timing will vary depending on the configuration.

Note: The results of these pre-release tests will not necessarily be those of the generally available C/400 product.

1. COBOL WRITE is somewhat faster than C/400 fwrite() to a physical file (without logical files attaching to it).

   Since no logical files are defined on the physical file, the speed will depend solely on how fast the COBOL WRITE and C/400 fwrite() can write a record to AS/400. In COBOL/400, WRITE is a statement of the language while in C/400, fwrite() is a function that will be called to write the record. COBOL or RPG write should be faster than C/400's fwrite().

2. As more logical files are defined across the physical file the COBOL WRITE and C/400 fwrite() are about the same speed.

   The test started with a physical file without a logical file attaching to it. In second run, one logical file was defined to a physical file to repeat the test. Finally, a second logical file was added and the test was repeated.

   With a stand alone physical file, COBOL WRITE is much faster than C/400 fwrite(), but as more logical files are added, the time required to write the same number of records to file is almost the same. We believe this is due to the time required to maintain the logical files. As the number of logical files increase, eventually this becomes the major portion of the time in the write process.

3. When writing to an AS/400 database file, using QXXFORMAT with fwrite() is slightly slower than a straightforward fwrite().

   QXXFORMAT is a function call and will require overhead. The test indicates that a fwrite() with QXXFORMAT requires about 20% more time than a straightforward fwrite().

4. For random record read and without OPNDBF, C/400 with SQL is faster than C/400 calling COBOL to read a record. C/400 with SQL is about 50% to 70% of the time required by C/400 calling COBOL to read a record.

   There are two kinds of overhead associated with C/400 calling a COBOL to read a record. The first one is loading the COBOL program, the second is file opening (in this test OPNDBF was not used). Therefore C/400 calling a COBOL program to read a record is slower than C/400 with SQL. Since the COBOL program is called repeatedly, the program probably stayed in memory most of the time and the time difference was probably due to the overhead of opening the file.

5. For random record read and with OPNDBF, C/400 with SQL is slower than C/400 calling COBOL to read a record. C/400 with SQL is considerably slower than C/400 calling a COBOL to read record.

   In this test, OPNDBF was entered before the programs are executed. Overhead due to opening the file in COBOL was reduced to minimum. Since COBOL read is faster than SQL read and not much overhead involved in opening the file in COBOL, C/400 calling COBOL to read record is faster than C/400 with SQL.

6. If a COBOL program is called repeatedly to read records, OPNDBF can significantly improve the speed. C/400 calling COBOL with OPNDBF is faster than C/400 calling COBOL without OPNDBF.

7. COBOL READ is the fastest way to read a record when compared with C/400 calling COBOL, C/400 with embedded SQL or COBOL with SQL.

   The test shows that COBOL read is faster than C/400 calling COBOL to read and considerably faster than C with SQL. COBOL with embedded SQL read was in between the first two times.

8. C/400 binary calculation is considerably faster than COBOL with zoned decimal.

   In all the tests that compare arithmetic operation under C/400 versus COBOL it was found that as the number of calculations increased the arithmetic operation under C/400 out-performed COBOL more and more. One reason to explain this is that starting a C/400 environment when a C/400 is called takes time. When there is only limited calculation, the C/400 start up time may be a dominant part of the total time. As the number of calculations increases, the C/400 start up time becomes a small portion of the total time.

9. C/400 binary calculation is also considerably faster than COBOL with packed decimal.

10. C/400 binary calculation is many factors faster than COBOL with binary numbers.

11. C/400 binary calculation is slightly faster than C/400 floating point.

12. COBOL calculation with packed field is slightly faster than zoned decimal.

    The AS/400 performs its internal calculations in packed data and thus the zoned-to-packed conversion will slow the program.

13. COBOL character string move is faster than C/400 strncpy.

14. C/400 memcpy was considerably faster than strncpy.

6.2 Performance Observations from Coding Analysis Tests

As in most other languages C/400 offers a flexible approach to coding. There are often many ways to code a program to obtain the same results. When coding a program, if there are two choices available to a programmer it would be to our advantage to use the most efficient coding techniques. The following observations were made from a Construct Analysis performed at the Toronto Laboratory. They may be used to help in the development of a more efficient C/400 program.

Note: The results of these pre-release tests will not necessarily be those of the generally available C/400 product.

The following provides a high level summary of the analysis that this testing has shown. More details from this Construct Analysis will be available in a HONE item later. The topic will be titled *C/400 Tips and Techniques.*

1. It was observed that the performance of an application improves if the array or vector declarations are kept together and at the end of the data declarations. This is especially true if the size of the array or vector is large.
2. The use of prototyping is encouraged as good programming technique. However, it has also been found to improve performance.
3. For improved compile time include only the header files for the C library routines used. For example if a program uses the strcmp function then include the header file string.h. However, only include the necessary header files. For example if the math functions are not used, then do not include the math.h header file.
4. When compiling or executing a C program ensure that the pool size of the subsystem is not too low. For example it was found that by changing the pool size from 500K to 1000K the execution and compile time improved quite significantly. Adequacy of the pool size varies for the program.
5. When using library routines/functions avoid the use of excess parenthesis. For example the performance of a statement with "strcmp (.....)" will be better than one with "(strcmp (.....))".
6. Use of casting impacts the performance negatively. Try to avoid mixed arithmetic so that casting can be avoided. For example avoid the use of double precision variables and single precision variables in a single statement.
7. If convenient avoid the use of the DATA CONVERSION routines, for handling zoned and packed data types. Try to use C/400 supported data types ie. float, character, double, integer and short.
8. It was observed that the performance of

   ```
   integer = integer +(or-or/or*) float
   ```

   was better than

   ```
   integer = float + (or-or/or*) integer.
   ```

   Or in other words, in the case of mixed arithmetic, it is advisable to make the first variable on the right hand side of an equation of the same type as the left hand side.
9. Use of pointers for accessing vectors does not help in performance. However, arrays should be accessed through pointers, if possible, to improve performance.
10. If you need to pass data to a C/400 program from COBOL then use parameters instead of calling the C program from the COBOL program.
11. During the pre-release testing the use of memory string handling routines were found to be the most efficient string handling routines. For example memcpy out performed strcpy.
12. Use of SETPGMINF command improves performance, particularly if program requires a large data area. In this command there are options to place data

areas within PAG or out of PAG. If program variables can be accommodated within PAG, execution of the program improves.

13. Use the Application Performance Tuning Aid when appropriate. It will provide better performance in the case of most call intensive applications. This is not a global recommendation for all applications since the size may become too large and cause excessive chaining and paging. Thus it is advisable to read the *Application Performance Tuning Aid* manual before using the product.
14. Stream I/O does not perform well on AS/400. Record I/O for sequential read and write performs very well. Therefore, for I/O operations use of Record I/O is recommended.
15. If COBOL is required to perform keyed/relative I/O then have an initial COBOL program to set up the environment and then call the C main program from this dummy COBOL program.
16. Use record I/O with a block parameter and follow the alignment rules for the definition of structures.
17. Use of fprintf or sprintf is expensive. This means that the cost of developing report producing application will be high if using C/400. Therefore, if this is the purpose of the application it will likely be most efficient to use COBOL or RPG.

6.3 Test programs

Test programs are used to compare the speed of performing a function using C/400 versus doing the same thing with COBOL and SQL.

All test programs were called from the main program. The main program shows a screen requesting the operator to type in descriptive text. The start time will be written to a file and the test program is called. When the test program completes its execution, the control is returned to the main program which recorded the end-time in the file.

List of Test Programs

TEST01 - C/400 program to write record to file with fwrite
TEST02 - COBOL program to write record to file
TEST03 - C/400 program to write record to file with QXXFORMAT
TEST04 - C/400 program to write record to file with fwrite and move fields
TEST05 - COBOL program to write record to file and move fields

TEST06 - C/400 program on arithmetic operation with binary (integer)
TEST07 - COBOL program on arithmetic operation with zone decimal
TEST08 - C/400 program on arithmetic operation with floating point
TEST09 - COBOL program on arithmetic operation with binary
TEST10 - COBOL program on arithmetic operation with pack decimal

TEST11 - C/400 program call COBOL for random record read
TEST12 - C/400 program with SQL for random record read
TEST13 - COBOL program with random record read
TEST14 - COBOL program with SQL for random record read

TEST15 - C/400 program with strncpy
TEST16 - C/400 program with memcpy

6.4 Main Program

The main program will display a screen to the operator requesting the text description. Once the operator presses the Enter key, the start time will be recorded in a record in a file. The corresponding test program will be called. After the test program completes execution, control is passed back to the main program which records the end time to a record in the file.

In this test, the main program was altered for an individual test program by changing one statement in the program.

Display File of Main Program - TSCREEN

```
A*  89/06/14  13:05:50    DAVID       REL-R02M00  5728-PW1
A                                      DSPSIZ(24 80 *DS3)
A          R FORMAT1
A*  89/06/14  13:05:50    DAVID       REL-R02M00  5728-PW1
A                                      CA03(01 'End of job')
A                                      BLINK
A                                    3 27'Test program running'
A                                      DSPATR(HI)
A                                      DSPATR(UL)
A                                    6 10'Program Name :'
A            PGMNAME       10   B  6 27
A                                    9 10'Number :'
A            PGMNUM         6  0B  9 21
A                                   12  9'Remark :'
A            REMARK        50   B 12 20
A                                   18  6'Enter <E> or F-3 to End =>'
A            COPTION        1   B 18 35
A            ERRORMSG      60   O 21  7DSPATR(HI)
A                                      DSPATR(BL)
```

Database File Used by Main Program - FSCREEN

```
A          R FSCREENF
A            PGMNAME       10         COLHDG('PGM NAME')
A            PGMNUM         6  0       COLHDG('NUMBER')
A            TSTART         8         COLHDG('START')
A            TSTOP          8         COLHDG('STOP')
A            REMARK        50         COLHDG('REMARK')
```

COBOL Main Program

```
 PROCESS APOST.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SAM00.
* ---- CHANGE PROGRAM ID. ACCORDINGLY TO INDIVIDUAL TEST
*      PROGRAM
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-AS400.
 OBJECT-COMPUTER. IBM-AS400.
*
 INPUT-OUTPUT SECTION.
```

```
 FILE-CONTROL.
     SELECT MFILE ASSIGN TO DATABASE-FSCREEN
         ORGANIZATION IS SEQUENTIAL
         ACCESS MODE IS SEQUENTIAL.
*
     SELECT DSPFILE ASSIGN TO WORKSTATION-TSCREEN
         ORGANIZATION IS TRANSACTION.
*
 DATA DIVISION.
 FILE SECTION.
 FD  MFILE LABEL RECORDS ARE STANDARD.
 01  MFILE-RECORD.
     COPY DDS-ALL-FORMAT OF FSCREEN.
*
 FD  DSPFILE LABEL RECORDS ARE OMITTED.
 01  DSPREC PIC X(200).
*
 WORKING-STORAGE SECTION.
 01  SCREEN-IN.
     COPY DDS-FORMAT1-I OF TSCREEN.
*
 01  SCREEN-OUT.
     COPY DDS-FORMAT1-O OF TSCREEN.
*
 01  INDICATOR-AREA.
     05  IN01 PIC 1.
         88  YES-END-OF-JOB VALUE B'1'.
         88  NOT-END-OF-JOB VALUE B'0'.
 77  ERROR-MESSAGE PIC X(60) VALUE
         'ERROR : INVALID CUSTOMER NUMBER ENTERED'.
 77  PROGRAM-NAME PIC X(10).
 77  OLD-REMARK   PIC X(70).
 77  OLD-NUMBER   PIC 9(6).
 01  PARAMETER-ITEM.
     05 THE-NUMBER  PIC X(6).
     05 P1NULL      PIC 9 COMP-4 VALUE 0.
*
*
 PROCEDURE DIVISION.
 BEGIN.
     OPEN EXTEND MFILE I-O DSPFILE.
     MOVE ALL SPACES TO FORMAT1-O.
     MOVE ZERO TO PGMNUM OF FORMAT1-O.
     WRITE DSPREC FROM SCREEN-OUT FORMAT 'FORMAT1'.
     PERFORM LOOP UNTIL COPTION OF FORMAT1-I = 'E'.
     CLOSE MFILE DSPFILE.
     STOP RUN.
 LOOP.
     MOVE SPACES TO ERRORMSG.
     READ DSPFILE INTO SCREEN-IN FORMAT 'FORMAT1'.
     MOVE CORR FORMAT1-I-INDIC TO INDICATOR-AREA.
     IF YES-END-OF-JOB
        MOVE 'E' TO COPTION OF FORMAT1-I.
     IF COPTION OF FORMAT1-I NOT = 'E' THEN
         PERFORM READ-LOOP THRU READ-LOOP-EXIT.
 READ-LOOP.
     MOVE CORR FORMAT1-I TO FORMAT1-O.
     IF PGMNUM OF FORMAT1-I = SPACES THEN
         MOVE 'ERROR : PROGRAM NAME CANNOT BE BLANK'
```

```
                TO ERRORMSG OF FORMAT1-O
                GO TO READ-LOOP-1.
           MOVE CORR FORMAT1-I TO FSCREENF.
           MOVE SPACES TO TSTOP OF FSCREENF.
           MOVE PGMNAME OF FORMAT1-I TO PROGRAM-NAME.
           MOVE REMARK OF FORMAT1-I TO OLD-REMARK.
           MOVE PGMNUM OF FORMAT1-I TO OLD-NUMBER.
           ACCEPT TSTART OF FSCREENF FROM TIME.
           WRITE MFILE-RECORD.
      *
      * ------------------------------------------------------------
      *
      *    Note that OLD-NUMER is number of times the Test Program
      *    will loop.
      *
      *    The Test Program name is specified in the CALL statement
      *    and will be changed for individual Test Program
      *
      *
      * ------------------------------------------------------------
           MOVE OLD-NUMBER TO THE-NUMBER.
           CALL "S22CPGA" USING OLD-NUMBER.
      *
      *
           MOVE SPACES TO TSTART OF FSCREENF.
           MOVE OLD-REMARK TO REMARK OF FSCREENF.
           MOVE OLD-NUMBER TO PGMNUM OF FSCREENF.
           MOVE PROGRAM-NAME TO PGMNAME OF FSCREENF.
           ACCEPT TSTOP OF FSCREENF FROM TIME.
           WRITE MFILE-RECORD.
       READ-LOOP-1.
           WRITE DSPREC FROM SCREEN-OUT FORMAT 'FORMAT1'.
       READ-LOOP-EXIT.
           EXIT.
```

6.5 TEST01 - C/400 Program to Write Record to File with fwrite

```
/*  Program Name : TEST01
    This program writes records to file with fwrite statement */

 #include <signal.h>
 #include <stdio.h>
 #include <errno.h>
 #include <string.h>
 #include <xxasio.h>
 #include <xxfdbk.h>
 #include <stdlib.h>

 #define SIZENUM  5
 #define SIZENAME 30
 #define SIZEADD  30
 #define SIZETEL  10
 #define CNUM "0001\0"
 #define CNAME "This is your name\0"
 #define CADD "This is your address\0"
 #define CTEL "Phone\0"
 #define FLD1 "This is field 1\0"
 #define FLD2 "This is field 2\0"
```

```
 #define FLD3 "This is field 3\0"
 #define CAMT 1234

main(int argc,char *argv[])
 {
 FILE *file1;
 char c;
 int i;
 int j;

struct frecord1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          char fld1[30];
          char fld2[30];
          char fld3[30];
          int camt;
          int num;
          }   frecord ;

    j=atoi(argv[1]);  /* j will be number of times to loop */

    file1=fopen("FSAM11","wb type=record");


        strcpy(frecord.cnum, CNUM);
        strcpy(frecord.cname, CNAME);
        strcpy(frecord.cadd, CADD);
        strcpy(frecord.ctel, CTEL);
        strcpy(frecord.fld1, FLD1);
        strcpy(frecord.fld2, FLD2);
        strcpy(frecord.fld3, FLD3);
        frecord.camt = CAMT;        /* copy constants into record
                                       buffer once               */

        for(i=0;i<j;i++)
        {
        frecord.num=i;
        fwrite(&frecord,sizeof(frecord),1,file1);  /* write record */
        }

    fclose(file1);

    exit;
}
```

6.6 TEST02 - COBOL Program to Write Record to File

Program TEST02 is a standard COBOL/400 program to write a record to the database file.

```
       PROCESS APOST.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST02.
      *
      *
```

```
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-AS400.
 OBJECT-COMPUTER. IBM-AS400.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT MFILE ASSIGN TO DATABASE-FSAM11A
         ORGANIZATION IS SEQUENTIAL
         ACCESS MODE IS SEQUENTIAL.
*
*
 DATA DIVISION.
 FILE SECTION.
 FD  MFILE LABEL RECORDS ARE STANDARD.
 01  MFILE-RECORD.
     COPY DDS-ALL-FORMAT OF FSAM11A.
*
*
 WORKING-STORAGE SECTION.
 77  COUNT-NUMBER  PIC 9(6).
 77  ITEM PIC 9(6).
 01  MRECORD.
     05  CNUM  PIC X(5).
     05  CNAME PIC X(30).
     05  CADD  PIC X(30).
     05  CTEL  PIC X(10).
     05  FLD1  PIC X(30).
     05  FLD2  PIC X(30).
     05  FLD3  PIC X(30).
     05  CAMT  PIC S9(8) COMP-3.
     05  NUM   PIC S9(6) COMP-3.
*
 LINKAGE SECTION.
 01  PARAMETER-ITEM  PIC X(6).
*
*
 PROCEDURE DIVISION USING PARAMETER-ITEM.
 BEGIN.
     OPEN OUTPUT MFILE.
     MOVE PARAMETER-ITEM TO ITEM.
     MOVE '00001' TO CNUM OF MRECORD.
     MOVE 'THIS IS YOUR NAME' TO CNAME OF MRECORD.
     MOVE 'THIS IS YOUR ADDRESS' TO CADD OF MRECORD.
     MOVE 'PHONE' TO CTEL OF MRECORD.
     MOVE 'THIS IS FIELD 1' TO FLD1 OF MRECORD.
     MOVE 'THIS IS FIELD 2' TO FLD2 OF MRECORD.
     MOVE 'THIS IS FIELD 3' TO FLD3 OF MRECORD.
     MOVE 1234 TO CAMT OF MRECORD.
     MOVE 1 TO COUNT-NUMBER.
     PERFORM LOOP UNTIL COUNT-NUMBER = ITEM.
     CLOSE MFILE.
     EXIT PROGRAM.
 LOOP.
     MOVE COUNT-NUMBER TO NUM OF MRECORD.
     WRITE MFILE-RECORD FROM MRECORD.
     COMPUTE COUNT-NUMBER = COUNT-NUMBER + 1.
```

DDS for Database File FSAM11A

```
A          R FSAM11AF
A            CNUM          5          COLHDG('NUMBER')
A            CNAME        30          COLHDG('NAME')
A            CADD         30          COLHDG('ADDRESS')
A            CTEL         10          COLHDG('PHONE')
A            FLD1         30          COLHDG('FLD1')
A            FLD2         30          COLHDG('FLD2')
A            FLD3         30          COLHDG('FLD3')
A            CAMT          8  0       COLHDG('AMOUNT')
A            NUM           6  0       COLHDG('NUM')
```

6.7 TEST03 - C/400 Program to Write Record to File with QXXFORMAT

```
/* Program Name : TEST03

This program writes records to a file. Before fwrite() is executed,
   QXXFORMAT is used to specify the record format.              */

 #include <signal.h>
 #include <stdio.h>
 #include <errno.h>
 #include <string.h>
 #include <xxasio.h>
 #include <xxfdbk.h>
 #include <stdlib.h>

 #define SIZENUM  5
 #define SIZENAME 30
 #define SIZEADD  30
 #define SIZETEL  10
 #define CNUM "0001\0"
 #define CNAME "This is your name\0"
 #define CADD "This is your address\0"
 #define CTEL "Phone\0"
 #define FLD1 "This is field 1\0"
 #define FLD2 "This is field 2\0"
 #define FLD3 "This is field 3\0"
 #define CAMT "1234"
 #define NUM "5678"

main(int argc,char *argv[])
 {
 FILE *file1;
 char c;
 int i;
 int j;

struct frecord1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          char fld1[30];
          char fld2[30];
          char fld3[30];
```

```
        char camt[6];
        char num[6];
        }   frecord ;

    j=atoi(argv[1]);

    file1=fopen("FSAM11B","wb type=record");

        strcpy(frecord.cnum, CNUM);
        strcpy(frecord.cname, CNAME);
        strcpy(frecord.cadd, CADD);
        strcpy(frecord.ctel, CTEL);
        strcpy(frecord.fld1, FLD1);
        strcpy(frecord.fld2, FLD2);
        strcpy(frecord.fld3, FLD3);
        strcpy(frecord.camt, CAMT);
        strcpy(frecord.num, NUM);

        for(i=0;i<j;i++)
        {
        QXXFORMAT(file1,"FSAM11BF  ");
        fwrite(&frecord,sizeof(frecord),1,file1);
        }

    fclose(file1);

    exit;
}
```

DDS for Database File FSAM11B

```
     A          R FSAM11BF
     A            CNUM          5         COLHDG('NUMBER')
     A            CNAME        30         COLHDG('NAME')
     A            CADD         30         COLHDG('ADDRESS')
     A            CTEL         10         COLHDG('PHONE')
     A            FLD1         30         COLHDG('FLD1')
     A            FLD2         30         COLHDG('FLD2')
     A            FLD3         30         COLHDG('FLD3')
     A            CAMT          6         COLHDG('AMOUNT')
     A            NUM           6         COLHDG('NUMBER')
```

6.8 TEST04 - C/400 Program to Write Record to File with fwrite and Move Fields

```
/* Program Name : TEST04

This program is similar to TEST01 except each time the fwrite()
   loop is performed, a series of strncpy is used to copy fields
   into output buffer at the same time.
                                                                  */

 #include <signal.h>
 #include <stdio.h>
 #include <errno.h>
 #include <string.h>
 #include <xxasio.h>
 #include <xxfdbk.h>
```

```
#include <stdlib.h>

#define SIZENUM  5
#define SIZENAME 30
#define SIZEADD  30
#define SIZETEL  10
#define CNUM "0001\0"
#define CNAME "This is your name\0"
#define CADD "This is your address\0"
#define CTEL "Phone\0"
#define FLD1 "This is field 1\0"
#define FLD2 "This is field 2\0"
#define FLD3 "This is field 3\0"
#define CAMT 1234

main(int argc,char *argv[])
 {
 FILE *file1;
 char c;
 int i;
 int j;

struct frecord1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          char fld1[30];
          char fld2[30];
          char fld3[30];
          int camt;
          int num;
          }   frecord ;

    j=atoi(argv[1]);  /* j will be number of times to loop */

    file1=fopen("FSAM11","wb type=record");

        for(i=0;i<j;i++)
        {
        strcpy(frecord.cnum, CNUM);
        strcpy(frecord.cname, CNAME);
        strcpy(frecord.cadd, CADD);
        strcpy(frecord.ctel, CTEL);
        strcpy(frecord.fld1, FLD1);
        strcpy(frecord.fld2, FLD2);
        strcpy(frecord.fld3, FLD3);
        frecord.camt = CAMT;        /* copy constants into record */

        frecord.num=i;
        fwrite(&frecord,sizeof(frecord),1,file1);  /* write record */
        }

    fclose(file1);

    exit;
}
```

6.9 TEST05 - COBOL Program to Write Record to File and Move Fields

```
Program Name : TEST05

This program is similar to TEST02 except that in each loop of
  WRITE, constants are moved into the output record buffer at the
  same time.

      PROCESS APOST.
      IDENTIFICATION DIVISION.
      PROGRAM-ID. TEST05.
     *
     *
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SOURCE-COMPUTER. IBM-AS400.
      OBJECT-COMPUTER. IBM-AS400.
     *
      INPUT-OUTPUT SECTION.
      FILE-CONTROL.
          SELECT MFILE ASSIGN TO DATABASE-FSAM11A
              ORGANIZATION IS SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.
     *
     *
      DATA DIVISION.
      FILE SECTION.
      FD  MFILE LABEL RECORDS ARE STANDARD.
      01  MFILE-RECORD.
          COPY DDS-ALL-FORMAT OF FSAM11A.
     *
     *
      WORKING-STORAGE SECTION.
      77  COUNT-NUMBER  PIC 9(6).
      77  ITEM PIC 9(6).
      01  MRECORD.
          05  CNUM  PIC X(5).
          05  CNAME PIC X(30).
          05  CADD  PIC X(30).
          05  CTEL  PIC X(10).
          05  FLD1  PIC X(30).
          05  FLD2  PIC X(30).
          05  FLD3  PIC X(30).
          05  CAMT  PIC S9(8) COMP-3.
          05  NUM   PIC S9(6) COMP-3.
     *
      LINKAGE SECTION.
      01  PARAMETER-ITEM  PIC X(6).
     *
     *
      PROCEDURE DIVISION USING PARAMETER-ITEM.
      BEGIN.
          OPEN OUTPUT MFILE.
          MOVE PARAMETER-ITEM TO ITEM.
          MOVE 1 TO COUNT-NUMBER.
          PERFORM LOOP UNTIL COUNT-NUMBER = ITEM.
          CLOSE MFILE.
```

```
            EXIT PROGRAM.
        LOOP.
            MOVE COUNT-NUMBER TO NUM OF MRECORD.
            MOVE '00001' TO CNUM OF MRECORD.
            MOVE 'THIS IS YOUR NAME' TO CNAME OF MRECORD.
            MOVE 'THIS IS YOUR ADDRESS' TO CADD OF MRECORD
            MOVE 'PHONE' TO CTEL OF MRECORD.
            MOVE 'THIS IS FIELD 1' TO FLD1 OF MRECORD.
            MOVE 'THIS IS FIELD 2' TO FLD2 OF MRECORD.
            MOVE 'THIS IS FIELD 3' TO FLD3 OF MRECORD.
            MOVE 1234 TO CAMT OF MRECORD.
            WRITE MFILE-RECORD FROM MRECORD.
            COMPUTE COUNT-NUMBER = COUNT-NUMBER + 1.
```

6.10 TEST06 - C/400 Program on Arithmetic Operation with Binary (Integer)

```
/* Program Name : TEST06

Each loop performs 4 arithmetic operations :
         - one addition
         - one subtraction
         - one multiplication
         - one division
    Note that C/400 use binary for data type integer        */

 #include <signal.h>
 #include <stdio.h>
 #include <errno.h>
 #include <string.h>
 #include <xxasio.h>
 #include <xxfdbk.h>
 #include <stdlib.h>

main(int argc,char *argv[])
 {
 char c;
 int i;
 int j;
 int a;
 int b;
 int c1;
 int d;
 int e;
 int f;
 int g;
 int h;

  a=3;
  b=4;
  c1=5;
  d=6;

    j=atoi(argv[1]);

        for(i=0;i<j;i++)
        {
        e=a+b;
```

```
        f=b-a;
        g=c1*d;
        h=d/c1;
        }
    exit;
}
```

6.11 TEST07 - COBOL Program on Arithmetic Operation with Zone Decimal

```
Program Name : TEST07

Each loop performs 4 arithmetic operations :
         - one addition
         - one subtraction
         - one multiplication
         - one division

       PROCESS APOST.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST07.
      *
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-AS400.
       OBJECT-COMPUTER. IBM-AS400.
      *
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
      *
      *
       DATA DIVISION.
       FILE SECTION.
      *
      *
       WORKING-STORAGE SECTION.
       77  COUNT-NUMBER  PIC 9(6).
       77  ITEM PIC 9(6).
       77  A PIC 999.
       77  B PIC 999.
       77  C PIC 999.
       77  D PIC 999.
       77  E PIC 999.
       77  F PIC 999.
       77  G PIC 999.
       77  H PIC 999.
      *
       LINKAGE SECTION.
       01  PARAMETER-ITEM  PIC X(6).
      *
      *
       PROCEDURE DIVISION USING PARAMETER-ITEM.
       BEGIN.
           MOVE PARAMETER-ITEM TO ITEM.
           MOVE 3 TO A.
           MOVE 4 TO B.
           MOVE 5 TO C.
```

```
           MOVE 6 TO D.
           MOVE 1 TO COUNT-NUMBER.
           PERFORM LOOP UNTIL COUNT-NUMBER = ITEM.
           EXIT PROGRAM.
       LOOP.
           COMPUTE E = A + B.
           COMPUTE F = B - A.
           COMPUTE G = C * D.
           COMPUTE H ROUNDED = D / C.
           COMPUTE COUNT-NUMBER = COUNT-NUMBER + 1.
```

6.12 TEST08 - C/400 Program on Arithmetic Operation with Floating Point

```
/* Program Name : TEST08

Each loop performs 4 arithmetic operations :
         - one addition
         - one subtraction
         - one multiplication
         - one division
 Note : that floating point is defined in the program       */

 #include <signal.h>
 #include <stdio.h>
 #include <errno.h>
 #include <string.h>
 #include <xxasio.h>
 #include <xxfdbk.h>
 #include <stdlib.h>

main(int argc,char *argv[])
 {
 char c;
 float i;
 float j;
 float a;
 float b;
 float c1;
 float d;
 float e;
 float f;
 float g;
 float h;

  a=3.3;
  b=4.4;
  c1=5.5;
  d=6.6;

    j=atoi(argv[1]);

        for(i=0;i<j;i++)
        {
        e=a+b;
        f=b-a;
        g=c1*d;
        h=d/c1;
```

```
        }
    exit;
}
```

6.13 TEST09 - COBOL Program on Arithmetic Operation with Binary

```
Program Name : TEST09

Each loop performs 4 arithmetic operations :
        - one addition
        - one subtraction
        - one multiplication
        - one division
 Note : that binary field is defined in the program

       PROCESS APOST.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST09.
      *
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-AS400.
       OBJECT-COMPUTER. IBM-AS400.
      *
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
      *
      *
       DATA DIVISION.
       FILE SECTION.
      *
      *
       WORKING-STORAGE SECTION.
       77  COUNT-NUMBER  PIC 9(6).
       77  ITEM PIC 9(6).
       77  A PIC 9999999V999 COMP-4.
       77  B PIC 9999999V999 COMP-4.
       77  C PIC 9999999V999 COMP-4.
       77  D PIC 9999999V999 COMP-4.
       77  E PIC 9999999V999 COMP-4.
       77  F PIC 9999999V999 COMP-4.
       77  G PIC 9999999V999 COMP-4.
       77  H PIC 9999999V999 COMP-4.
      *
       LINKAGE SECTION.
       01  PARAMETER-ITEM  PIC X(6).
      *
      *
       PROCEDURE DIVISION USING PARAMETER-ITEM.
       BEGIN.
           MOVE PARAMETER-ITEM TO ITEM.
           MOVE 3.3 TO A.
           MOVE 4.4 TO B.
           MOVE 5.5 TO C.
           MOVE 6.6 TO D.
           MOVE 1 TO COUNT-NUMBER.
           PERFORM LOOP UNTIL COUNT-NUMBER = ITEM.
```

```
            EXIT PROGRAM.
        LOOP.
            COMPUTE E = A + B.
            COMPUTE F = B - A.
            COMPUTE G = C * D.
            COMPUTE H ROUNDED = D / C.
            COMPUTE COUNT-NUMBER = COUNT-NUMBER + 1.
```

6.14 TEST10 - COBOL Program on Arithmetic Operation with Pack Decimal

```
Program Name : TEST10

Each loop performs 4 arithmetic operations :
         - one addition
         - one subtraction
         - one multiplication
         - one division
Note : that packed numeric fields are used in program

       PROCESS APOST.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST10.
      *
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-AS400.
       OBJECT-COMPUTER. IBM-AS400.
      *
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
      *
      *
       DATA DIVISION.
       FILE SECTION.
      *
      *
       WORKING-STORAGE SECTION.
       77  COUNT-NUMBER  PIC 9(6).
       77  ITEM PIC 9(6).
       77  A PIC 999 COMP-3.
       77  B PIC 999 COMP-3.
       77  C PIC 999 COMP-3.
       77  D PIC 999 COMP-3.
       77  E PIC 999 COMP-3.
       77  F PIC 999 COMP-3.
       77  G PIC 999 COMP-3.
       77  H PIC 999 COMP-3.
      *
       LINKAGE SECTION.
       01  PARAMETER-ITEM  PIC X(6).
      *
      *
       PROCEDURE DIVISION USING PARAMETER-ITEM.
       BEGIN.
           MOVE PARAMETER-ITEM TO ITEM.
           MOVE 3 TO A.
```

```
            MOVE 4 TO B.
            MOVE 5 TO C.
            MOVE 6 TO D.
            MOVE 1 TO COUNT-NUMBER.
            PERFORM LOOP UNTIL COUNT-NUMBER = ITEM.
            EXIT PROGRAM.
        LOOP.
            COMPUTE E = A + B.
            COMPUTE F = B - A.
            COMPUTE G = C * D.
            COMPUTE H ROUNDED = D / C.
            COMPUTE COUNT-NUMBER = COUNT-NUMBER + 1.
```

6.15 TEST11 - C/400 Program call COBOL for Random Record Read

This sample contains one C/400 program and one COBOL program. Function of program is to retrieve a database record based on the key passed from the C/400 program.

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

#define MAX 50000
#define SIZEKEY 6

#pragma linkage(S22CB1,OS)   /* S22CB1 is cbl pgm to read record */
extern void S22CB1(char *);

main(int argc,char *argv[])
{
   struct frecord
    {
     char ckey[6];
     char cnum[6];
     char fld1[30];
     char fld2[30];
     char fld3[30];
     char fld4[30];
     } frecord;

    int i;
    int j;
    int k;
    char s[6];
    char c;
    char xnum[8];
    char xnum1[6];
    char xnum2[7];

    strcpy(s,argv[1]);

    i=0;
    j=atoi(argv[1]);
```

```
    for (i=0;i<j;i++)
    {
        k=(rand() % MAX);    /* generate random number */

        strncpy(xnum,"        ",8);
        sprintf(xnum,"%d",k);  /* convert integer to alpha */

        strncpy(xnum1,xnum,6);
        xnum1[6]='\0';

        padnum(&xnum1,6);     /* right justify field and fill
                                 leading space with 0 */

        strncpy(frecord.ckey,xnum1,6);  /* move field to key */

        clrbuf(&frecord.fld1,30);
        clrbuf(&frecord.fld2,30);
        clrbuf(&frecord.fld3,30);
        clrbuf(&frecord.fld4,30);
        frecord.fld4[30]='\0';  /* clear record buffer */

        S22CB1(frecord.ckey);   /* call cbl program to read record */

    }
    exit;
  }

padnum(buffer,l)  /* make field right justify and fill leading
                     spaces with 0                              */

char *buffer;
int l;

{
 int i;
 int j;
 int k;
 char s[6];

 i=atoi(buffer);

 sprintf(s,"%d",i);

  k=0;
  for(j=0;j<sizeof(s);j++)
  {
   if(!isdigit(s[j]))
    {
     k=k+1;
    }
  }
  for(j=0;j<k;j++)
  {
   buffer[j]='0';
   }
  for(j=k;j<sizeof(s);j++)
   {
    buffer[j]=s[j-k];
```

```
    }
}

clrbuf(buffer,l)

char *buffer;
int l;

{
 int i;

 for(i=0;i<l;i++)
 {
  buffer[i]=' ';
  }
}

filspace(buffer,l)

 char *buffer;
 int l;

{
 int i;
  for(i=0;i<l;i++)
  {
   if(!(isprint(buffer[i])))
    buffer[i]=' ';
    }
  }
```

COBOL Program Called to Read File

```
       PROCESS APOST.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. S22CB1.
      *
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-AS400.
       OBJECT-COMPUTER. IBM-AS400.
      *
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT MFILE ASSIGN TO DATABASE-BIGF1l
               ORGANIZATION IS INDEXED
               ACCESS MODE IS RANDOM
               RECORD KEY IS EXTERNALLY-DESCRIBED-KEY.
      *
      *
       DATA DIVISION.
       FILE SECTION.
       FD  MFILE LABEL RECORDS ARE STANDARD.
       01  MFILE-RECORD.
           COPY DDS-ALL-FORMAT OF BIGF1L.
      *
      *
       WORKING-STORAGE SECTION.
```

```
       77  DUMMY-FIELD PIC XXX  VALUE SPACES.
      *
       LINKAGE SECTION.
       01  PARAMETER-LIST.
           05  CKEY  PIC X(6).
           05  CNUM  PIC X(6).
           05  FLD1  PIC X(30).
           05  FLD2  PIC X(30).
           05  FLD3  PIC X(30).
           05  FLD4  PIC X(30).
      *
      *
       PROCEDURE DIVISION USING PARAMETER-LIST.
       BEGIN.
           OPEN INPUT MFILE.
           MOVE CKEY TO CNUM OF PARAMETER-LIST.
           MOVE CORR PARAMETER-LIST TO FORMAT1.
           READ MFILE INVALID KEY STOP RUN.
           MOVE CORR FORMAT1 TO PARAMETER-LIST.
           EXIT PROGRAM.
```

6.16 TEST12 - C/400 Program with SQL for Random Record Read

This C/400 program includes SQL statement to read file by random key.

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

#define MAX 50000
#define SIZEKEY 6
/*********************************************************************/
      EXEC SQL INCLUDE SQLCA;
/*********************************************************************/

main(int argc,char *argv[])
 {
   struct frecord
    {
     char ckey[6];
     char cnum[6];
     char fld1[30];
     char fld2[30];
     char fld3[30];
     char fld4[30];
     } frecord;

    int i;
    int j;
    int k;
    char s[6];
    char c;
    char xnum[8];
    char xnum1[6];
```

```
    char xnum2[7];

    strcpy(s,argv[1]);

    i=0;

    j=atoi(argv[1]);
    for (i=0;i<j;i++)
   {
       k=(rand() % MAX);

       strncpy(xnum,"        ",8);
       sprintf(xnum,"%d",k);

       strncpy(xnum1,xnum,6);
       xnum1[6]='\0';

       padnum(&xnum1,6);
       strncpy(frecord.ckey,xnum1,6);

       clrbuf(&frecord.fld1,30);
       clrbuf(&frecord.fld2,30);
       clrbuf(&frecord.fld3,30);
       clrbuf(&frecord.fld4,30);
       frecord.fld4[30]='\0';

       EXEC SQL
            SELECT *
            INTO :frecord.cnum, :frecord.fld1, :frecord.fld2,
                 :frecord.fld3, :frecord.fld4
            FROM BIGF1L
            WHERE CNUM = :frecord.ckey;

    }
    exit;
   }

padnum(buffer,l)

char *buffer;
int l;

{
 int i;
 int j;
 int k;
 char s[6];

 i=atoi(buffer);

 sprintf(s,"%d",i);

  k=0;
  for(j=0;j<sizeof(s);j++)
  {
   if(!isdigit(s[j]))
   {
     k=k+1;
    }
```

```
    }
    for(j=0;j<k;j++)
    {
     buffer[j]='0';
     }
    for(j=k;j<sizeof(s);j++)
     {
      buffer[j]=s[j-k];
      }
 }

clrbuf(buffer,l)

char *buffer;
int l;

{
 int i;

 for(i=0;i<l;i++
 {
  buffer[i]=' ';
  }
 }

filspace(buffer,l)

 char *buffer;
 int l;

{
 int i;
  for(i=0;i<l;i++)
  {
   if(!(isprint(buffer[i])))
    buffer[i]=' ';
    }
  }
```

6.17 TEST13 - COBOL Program with Random Record Read

This is a standard COBOL program to read file.

```
       PROCESS APOST.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST13.
      *
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-AS400.
       OBJECT-COMPUTER. IBM-AS400.
      *
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT MFILE ASSIGN TO DATABASE-BIGF11
               ORGANIZATION IS INDEXED
               ACCESS MODE IS RANDOM
```

```
                RECORD KEY IS EXTERNALLY-DESCRIBED-KEY.
      *
      *
       DATA DIVISION.
       FILE SECTION.
       FD  MFILE LABEL RECORDS ARE STANDARD.
       01  MFILE-RECORD.
           COPY DDS-ALL-FORMAT OF BIGF1L.
      *
      *
       WORKING-STORAGE SECTION.
       77  DUMMY-FIELD PIC XXX  VALUE SPACES.
       01  NOW-TIME.
           05  NOW-HOUR PIC XX.
           05  NOW-MINUTE PIC XX.
           05  NOW-SECOND PIC XX.
       77  WORK-SECOND  PIC 99.
       77  NEW-KEY1 PIC 999999.
       77  BASE-KEY PIC 999999 VALUE 0.
       77  INCREMENT PIC 9999 VALUE 1000.
       77  TOP-NUM PIC 99999 VALUE 49000.
       77  NEW-KEY2 PIC 999999.
       77  LOOP-COUNTER PIC 999999.
       77  LOOP-TOTAL   PIC 999999.
       01  INDICATOR-AREA.
           05  TEST-CODE PIC 1.
               88  YES-TEST-ONE  VALUE B'1'.
               88  YES-TEST-TWO  VALUE B'0'.
      *
       LINKAGE SECTION.
       01  PARAMETER-LIST.
           05  CKEY  PIC X(6).
      *
      *
       PROCEDURE DIVISION USING PARAMETER-LIST.
       BEGIN.
           OPEN INPUT MFILE.
      * --------------  SET INITIAL VALUE --------------
           SET YES-TEST-ONE TO TRUE.
           MOVE CKEY TO LOOP-TOTAL.
           MOVE BASE-KEY TO NEW-KEY2.
           MOVE ZERO TO LOOP-COUNTER.
           PERFORM LOOP THRU LOOP-EXIT UNTIL
              LOOP-TOTAL = LOOP-COUNTER.
           CLOSE MFILE.
           GOBACK.
       LOOP.
           IF YES-TEST-ONE PERFORM GET-KEY-ONE.
           IF YES-TEST-TWO PERFORM GET-KEY-TWO.
           IF YES-TEST-ONE THEN
               SET YES-TEST-TWO TO TRUE
               ELSE
               SET YES-TEST-ONE TO TRUE.
      * --------------  READ FILE --------------
           READ MFILE INVALID KEY STOP RUN.
           COMPUTE LOOP-COUNTER = LOOP-COUNTER + 1.
       LOOP-EXIT.
           EXIT.
      *
```

```
 GET-KEY-ONE.
     ACCEPT NOW-TIME FROM TIME.
     MOVE NOW-SECOND TO WORK-SECOND.
     COMPUTE NEW-KEY1 = WORK-SECOND * WORK-SECOND.
     MOVE NEW-KEY1 TO CNUM OF FORMAT1.
*
 GET-KEY-TWO.
     IF NEW-KEY2 >= TOP-NUM
         MOVE ZERO TO NEW-KEY2.
     COMPUTE NEW-KEY2 = NEW-KEY2 + INCREMENT.
     MOVE NEW-KEY2 TO CNUM OF FORMAT1.
```

Database File Used in Program BIGF1L

Logical file BIGF1L is based on physical file BIGF1 with CNUM as key :

```
A          R FORMAT1
A            CNUM           6          COLHDG('NUMBER')
A            FLD1           30         COLHDG('FLD1')
A            FLD2           30         COLHDG('FLD2')
A            FLD3           30         COLHDG('FLD3')
A            FLD4           30         COLHDG('FLD4')
```

6.18 TEST14 - COBOL Program with SQL for Random Record Read

This is a COBOL program with SQL to read file.

```
 PROCESS APOST.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TEST14.
*
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-AS400.
 OBJECT-COMPUTER. IBM-AS400.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT MFILE ASSIGN TO DATABASE-BIGF1l
         ORGANIZATION IS INDEXED
         ACCESS MODE IS RANDOM
         RECORD KEY IS EXTERNALLY-DESCRIBED-KEY.
*
*
 DATA DIVISION.
 FILE SECTION.
 FD  MFILE LABEL RECORDS ARE STANDARD.
 01  MFILE-RECORD.
     COPY DDS-ALL-FORMAT OF BIGF1L.
*
*
 WORKING-STORAGE SECTION.
     EXEC SQL
       INCLUDE SQLCA
     END-EXEC.
 77  DUMMY-FIELD PIC XXX  VALUE SPACES.
 01  NOW-TIME.
```

```
           05  NOW-HOUR PIC XX.
           05  NOW-MINUTE PIC XX.
           05  NOW-SECOND PIC XX.
       77  WORK-SECOND  PIC 99.
       77  NEW-KEY1 PIC 999999.
       77  BASE-KEY PIC 999999 VALUE 0.
       77  INCREMENT PIC 9999 VALUE 1000.
       77  TOP-NUM PIC 99999 VALUE 49000.
       77  NEW-KEY2 PIC 999999.
       77  LOOP-COUNTER PIC 999999.
       77  LOOP-TOTAL   PIC 999999.
       01  INDICATOR-AREA.
           05  TEST-CODE PIC 1.
               88  YES-TEST-ONE  VALUE B'1'.
               88  YES-TEST-TWO  VALUE B'0'.
      *
       LINKAGE SECTION.
       01  PARAMETER-LIST.
           05  CKEY  PIC X(6).
      *
      *
       PROCEDURE DIVISION USING PARAMETER-LIST.
       BEGIN.
           OPEN INPUT MFILE.
      * --------------  SET INITIAL VALUE --------------
           SET YES-TEST-ONE TO TRUE.
           MOVE CKEY TO LOOP-TOTAL.
           MOVE BASE-KEY TO NEW-KEY2.
           MOVE ZERO TO LOOP-COUNTER.
           PERFORM LOOP THRU LOOP-EXIT UNTIL
              LOOP-TOTAL = LOOP-COUNTER.
           CLOSE MFILE.
           GOBACK.
       LOOP.
           IF YES-TEST-ONE PERFORM GET-KEY-ONE.
           IF YES-TEST-TWO PERFORM GET-KEY-TWO.
           IF YES-TEST-ONE THEN
               SET YES-TEST-TWO TO TRUE
               ELSE
               SET YES-TEST-ONE TO TRUE.
      * --------------  READ FILE WITH SQL --------------
             EXEC SQL
             SELECT *
             INTO :FORMAT1
             FROM BIGF1L
             WHERE CNUM = :CNUM
             END-EXEC.
           COMPUTE LOOP-COUNTER = LOOP-COUNTER + 1.
       LOOP-EXIT.
           EXIT.
      *
       GET-KEY-ONE.
           ACCEPT NOW-TIME FROM TIME.
           MOVE NOW-SECOND TO WORK-SECOND.
           COMPUTE NEW-KEY1 = WORK-SECOND * WORK-SECOND.
           MOVE NEW-KEY1 TO CNUM OF FORMAT1.
      *
       GET-KEY-TWO.
           IF NEW-KEY2 >= TOP-NUM
```

```
           MOVE ZERO TO NEW-KEY2.
         COMPUTE NEW-KEY2 = NEW-KEY2 + INCREMENT.
         MOVE NEW-KEY2 TO CNUM OF FORMAT1.
```

TEST15 - C/400 Program with strncpy

This program copy string with strncpy

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main(int argc,char *argv[])
 {
    int i;
    int j;
    char s1[30];
    char s2[30];
    char s3[30];
    char s4[30];
    char t1[30];
    char t2[30];
    char t3[30];
    char t4[30];

    strcpy(s,argv[1]);

    i=0;

    j=atoi(argv[1]);

    strncpy(s1,"this is for string1",30);
    strncpy(s2,"this is for string2",30);
    strncpy(s3,"this is for string3",30);
    strncpy(s4,"this is for string4",30);

    for (i=0;i<j;i++)
   {

     strncpy(t1,s1,30);
     strncpy(t2,s2,30);
     strncpy(t3,s3,30);
     strncpy(t4,s4,30);
    }
    exit;
  }
```

6.19 TEST16 - C/400 Program with memcpy

```
This program copy string with strncpy

 #include <signal.h>
 #include <stdio.h>
 #include <errno.h>
 #include <string.h>
 #include <xxasio.h>
 #include <xxfdbk.h>
 #include <stdlib.h>

main(int argc,char *argv[])
 {
    int i;
    int j;
    char s1[30];
    char s2[30];
    char s3[30];
    char s4[30];
    char t1[30];
    char t2[30];
    char t3[30];
    char t4[30];

    strcpy(s,argv[1]);

    i=0;

    j=atoi(argv[1]);

    strncpy(s1,"this is for string1",30);
    strncpy(s2,"this is for string2",30);
    strncpy(s3,"this is for string3",30);
    strncpy(s4,"this is for string4",30);

    for (i=0;i<j;i++)
   {
     memcpy(t1,s1,30);
     memcpy(t2,s2,30);
     memcpy(t3,s3,30);
     memcpy(t4,s4,30);
    }
    exit;
 }
```

Appendix A. Sample C Programs

A.1 Sample Programs

C is a new programming language to the AS/400 developer. In many aspects, this programming language does not work the same way as other programming languages (for example COBOL or RPG). C/400 has its own way of handling characters, strings, numeric data, pointers etc. C/400 also has its own way of storing the data on AS/400 and the format is often different from the conventional AS/400 approach.

The purpose of attaching the sample programs is to help the user more easily identify some key characteristics of the C/400 language.

All the sample programs shown here are simple, short programs coded in C/400. The programs are not coded in any advanced technique and they should not be considered model programs but they will serve to explain some of the C/400 characteristics.

Note: that '\0' means HEX 00, '\n' is the new line character and both are one byte long.

A.2 Example 1 - PGM01 (String Substitution)

Some of the sample programs were developed on a PC and uploaded to the AS/400 using PC Support. In the upload process, the left square bracket [and the right square bracket] were lost and were replaced by HEX 00. To correct the source manually with SEU would be very time-consuming since every string defined in a C program will require square brackets.

PGM01 is used to scan the source program and replace the HEX 00 characters with appropriate square brackets.

A modification of this program could be used to translate any characters in a C file.

```
/* ------------------------------------------------------ -----

   Program Name : PGM01

   This program illustrates how to use C/400 to convert a source
   program which was originally developed on a PC. The original source
   program is uploaded from PC to AS/400 through PC Support. In this
   process, the left square bracket [ and the right square bracket ]
   were lost and were replaced by HEX 00.

   The program reads the member in the source file as a binary file.
   It checks every character in the record to see whether it is a
   HEX 00 character. If it is the first HEX 00, it will then be
   replaced by character [ (value of D as defined in #define
   statement). If it is the second HEX 00 in the record, it will be
   replaced by character ] (value of E as defined in #define
```

```
   statement). If the character is not HEX 00, it will remain
   unchanged in the output file.

   This program will be called by an AS/400 CL program. The CL
   program will have two OVRDBF statements. Each OVRDBF will define
   explicitly which member in which source file will be used as
   input file file1 and which one will be used as output file file2.

------------------------------------------------------------------ */

#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

#define D 173 /* decimal value of character [ */
#define E 189 /* decimal value of character ] */

#define RECLEN 92 /* length of record in source file. If length of
                     record in source file change, only value of
                     RECLEN need to be changed as all length
                     definition in the program is based on this value.
                   */

main()

{
  FILE *file1;    /* Input source file from which the source will
                     be converted. This file can be defined by
                     OVRDBF to specify which member will be used */

  FILE *file2;    /* Output source file to hold the output of the
                     converted source. This file can be defined by
                     OVRDBF to specify which member will be used */

  struct record
    {
     char  s[RECLEN];
     } record;  /* The structure record contains s as its only
                   element and it has length of 92, as defined by
                   RECLEN. This structure will serve as input
                   record buffer in the program */

  struct record1
    {
     char  s1[RECLEN];
     } record1;  /* The structure record1 contains s1 as its only
                    element and its length is 92. This record will
                    serve as output record buffer in the program */

  int swit;      /* swit will be used as a switch in the program */


  file1=fopen("FILE1","rb type=record");
                /* FILE1 is the input file for processing.
                   file1 is the file pointer pointing to FILE1
```

```
                    on AS/400. FILE1 can be redirected by using
                    OVRDBF in which the source file, library and
                    the source member can be redefined.
                    FILE1 is defined as a binary file and it will
                    be processed in record mode.
                  */

  file2=fopen("FILE2","wb+ type=record");
                  /* FILE2 is the output file for processing.
                     file2 is the file pointer pointing to FILE2
                     on AS/400. FILE2 can be redirected by using
                     OVRDBF in which the source file, library and
                     the source member can be redefined.
                     FILE2 is defined as a binary file and it wil
                     be processed in record mode.
                     In this particular example, the source file
                     already exists. Therefore even lrecl (record
                     length) value is not specified and FILE2 will
                     be a new member to the file. The record length
                     of this new member is automatically set to
                     92.
                   */

  while(!feof(file1))     /* When not end of file1  */
  {
   fread(&record,sizeof(record),1,file1);
                         /* read the record from file1 into record
                            buffer. The record buffer is the
                            structure defined by record.
                            Note that only the address (pointer) of
                            input buffer can be specified in the
                            fread statement and length of buffer
                            must be explicitly specified.
                          */
   swit=0;       /* swit is initialized to 0  before processing
                    each record */

   for(i=0;i<RECLEN;i++) /* scan through the whole string */
    {
     if((record.s[i]=='\0') && (swit==1))
                        /* if character found to be HEX 00 and
                           another HEX 00 has been encountered
                           before in the same record, then
                           replace this character with the value of
                           E (189 as defined in #define statement)
                         */
         {
          record1.s1[i]=E;
          }

     if((record.s[i]=='\0') && (swit==0))
                         /* if a character found to be HEX 00 and
                            it is the first time encountered in the
                            string (swit == 0), then replace this
                            character with the value of D (173 as
                            defined in the #define statement)
                          */
         {
           record1.s1[i]=D;
```

```
            swit=1;      /* swit is set to 1 to denote that first
                            HEX 00 has been detected */
           }

      if(record.s[i]!='\0')
                         /* if character is not HEX 00, the output
                            character will be same as input character
                         */
         {
         record1.s1[i]=record.s[i];
         }
       }


      if(!feof(file1))  /* this is to avoid duplicating the last
                           statement in the source file twice
                        */
      {
       fwrite(&record1,sizeof(record1),1,file2);
                        /* write the output record to disk. the
                           output buffer is stored in structure
                           record1.
                         */
      }
    }
    fclose(file1);
    fclose(file2);
}
```

A.3 Example 2 - PGM02 (Unsigned Packed Field to Signed Packed Field Conversion)

A byte in a packed field contains 2 digits. In AS/400, the last half byte of the packed field contains the sign and a hex value of D to represent a negative number and any other value represent a positive number. In some computers, the positive number can be stored as unsigned pack field in order to save storage space. The characteristics is that these unsigned pack field does not use the last half byte to represent the sign, but these fields cannot be processed by AS/400.

As an example, the number +1234 will be represented on AS/400 in Hex as:

```
024
13F
```

but with unsigned packed field, this will be represented as

```
13
24
```

The difference we can see here is that:

(1). Field lengths in terms of bytes can be different, depending upon whether the field contains odd-number or even-number of digit.

(2). As compared with unsigned pack field, the digits are shifted half-byte higher.

PGM02 illustrates step by step how the half-byte shift is performed under C. It make use of C Language's ability to treat a character as an integer for arithmetic operation.

The same differences will be noticed when using data from other computers where formats are not compatible with AS/400 (for example, the first half-byte represents a sign instead of the last half-byte, Hex D does not represent negative etc).

```
/* ------------------------------------------------ ----------------

   Program Name : PGM02

   This program is a step-by-step illustration of how to convert an
   unsigned packed field to a signed packed field that can be processed by
   AS/400.

   In some computer, in order to save storage space, packed fields are
   stored without a sign (i.e. they always represent positive numbers).
   AS/400 process packed fields with a sign and conversion is required in
   such a situation.

   This program uses C Language's ability to treat a character
   as an integer. The objective of the program is to
   make a "half-byte shift" to the left so that a sign can be
   inserted into the lower half byte after the last digit.

   This program is a very straight-forward step-by-step illustration
   of the process. For a practical application, such function will be
```

```
   processed in the form of a function.

-------------------------------------------------------------------- */

#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>
#include <math.h>

main()
{
 char s[2];    /*   First string of unsigned pack field  */
 char s1[2];   /*   Second string of unsigned pack field */
 char t[3];    /*   Signed pack field to contain value of s.
                   Note that length is 3 byte instead of 2 byte */
 char t1[3];   /*   Signed pack field to contain value of s1.
                   Note that length is 3 byte instead of 2 byte */

 int du;       /*   To store upper half byte of data */
 int dl;       /*   To store lower half byte of data */

 s[0]=1;
 s[1]=35;      /*   These two statements assign HEX 01 23 to s
                   so as to simulate an unsigned pack field of
                   value +123    */

 s1[0]=18;
 s1[1]=52;     /*   These two statements assign HEX 12 34 to s1
                   so as to simulate an unsigned pack field of
                   value +1234       */

 t[0]='\0';
 t[1]='\0';
 t[2]='\0';
 t1[0]='\0';
 t1[1]='\0';
 t1[2]='\0';   /*   Initial values of t and t1 are all set to
                   value HEX 00      */

 printf("value of s[0] is %d\n",s[0]);
 printf("value of s[1] is %d\n",s[1]);
              /*  Print the original value of s. In HEX
                  the string s should be 01 23
              */

 du=upper(s[0]); /* Store the higher half-byte of s[0] to
                    du by calling function upper  */

 dl=lower(s[0]); /* Store lower half-byte of s[0] to dl
                    by calling function lower     */

 t[0]=du;  /* Store du to lower half-byte of t[0], note that
              t[0] is the first of the 3 byte and the first
              digit of s has been shifted in t in this process */
```

```
 t[1]= 16 * dl; /* Move the original lower half-byte to higher
                   half-byte by multiplying the number by 16 */

 du=upper(s[1]);
 dl=lower(s[1]);
 t[1]=t[1] + du;
 t[2]= 16 * dl; /* Repeat the process with other digits in the
                              string s */

 t[2]=t[2] + 15; /* Put a HEX F into last half-byte of the string
                    t to represent a positive number */

 printf("value of t[0] is %d\n",t[0]);
 printf("value of t[1] is %d\n",t[1]);
 printf("value of t[2] is %d\n",t[2]);
 printf("\n");     /* Display the converted string. Its value should
                       be HEX 00 12 3F                          */

   /* ---------------------------------------------
      The following repeat the previous step on s1
      --------------------------------------------- */

 printf("value of s1[0] is %d\n",s1[0]);
 printf("value of s1[1] is %d\n",s1[1]);
         /*  The HEX value should be 12 34      */

 du=upper(s1[0]);
 dl=lower(s1[0]);
 t1[0]=du;
 t1[1]= 16 * dl;
 du=upper(s1[1]);
 dl=lower(s1[1]);
 t1[1]=t1[1] + du;
 t1[2]= 16 * dl;
 t1[2]=t1[2] + 15;

 printf("value of t1[0] is %d\n",t1[0]);
 printf("value of t1[1] is %d\n",t1[1]);
 printf("value of t1[2] is %d\n",t1[2]);
            /*  The converted value should be HEX 01 23 4F  */
}

lower(i)   /* function to obtain lower half-byte of a byte */

int i;

{
 char a;

 a=fmod(i,16);    /* a keep the remainder of i divided by 16 */

 return(a);       /* return the remainder to main function */
}

upper(i)  /*  function to obtain higher half-byte of the byte */

int i;

{
```

```
 char a;

 a=(i - fmod(i,16))/16;  /* a now contain upper half-byte of i */

 return(a);  /* return a to main function */
}
```

A.4 Example 3 - PGM03 (Writing Records to Program-Described File with C)

This is a simple example of writing records to a program-described file on AS/400. Note that the file is defined as a binary file and it must exist before the program is executed.

It is interesting to note that camt is defined as an integer field in the program, but when DSPPFM command is used to browse the physical file, we can see that camt is actually stored in binary form.

If we analyze the record layout in greater detail, we can also see the record length difference due to data alignment. The first field in the record is CNUM and is 5 character, second field is CNAME and is 30 characters, third field is CADD and is 30 characters, fourth field is CTEL and is 10 characters. Integer under C occupies 4 bytes, so by adding length of individual fields together, the record length should be 79 byte. But when sizeof() function is used to retrieve the length of the record, it shows the length should be 80. This is because integers always starts at a word boundary (multiple of 4 byte). When the first 4 fields in the records are added together, its length is 75, rounding to a word boundary, it is 76 byte, and adding 4 more bytes for integer, the record length is therefore 80.

If in this particular example, the physical file TFILEB is created with a field length of 79, errors will occur.

```
/* ------------------------------------------------------

   Program Name : PGM03 (writing records with C to program
                         described file)

   This is a simple program illustrating how to write records
   to a program-described file. It includes the typical
   statements that will be used in writing records with
   C/400.

------------------------------------------------------------  */

#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

#define SIZENUM  5
#define SIZENAME 30
#define SIZEADD  30
#define SIZETEL  10  /* Length of fields are defined here as
                        constants. The advantage of doing so
                        is that in case the field length has
                        to be changed, only these part of codes
                        needed to be changed and no need to touch
                        the other program codes  */
```

```
main()
{
 FILE *file1;   /* file1 is the output file   */
 char c;
 char amount[10];

struct frecord1 {
          char cnum[SIZENUM];
          char cname[SIZENAME];
          char cadd[SIZEADD];
          char ctel[SIZETEL];
          int camt;
          } frecord ; /* This structure will be defined as the
                         output record buffer */

    file1=fopen("TFILEB","wb+ type=record");
                        /* The file pointer file1 is pointing
                           to TFILEB in AS/400. It will be
                           processed in record mode as type=
                           record is specified. The file is
                           opened for write output (wb).
                           This file must exist before the
                           program runs. It can be created by
                           CRTPF command with record length
                           specified and without DDS.*/

       c=' ';
       while(c!='$')  /* Will loop until '$' is entered
                         from screen                    */
       {
          printf("Enter Number :\n");  /* prompting message */
          readfld(&frecord.cnum,SIZENUM);  /* read from screen
                                              by calling function
                                              readfld          */

          printf("Enter Name :\n");
          readfld(&frecord.cname,SIZENAME);

          printf("Enter Address :\n");
          readfld(&frecord.cadd,SIZEADD);

          printf("Enter Phone :\n");
          readfld(&frecord.ctel,SIZETEL);

          printf("Enter Amount :\n");
          readfld(&amount,sizeof(amount));
          frecord.camt = atoi(amount); /* amount is first
                read as string then converted to numeric
                by function atoi. The reason for doing so is
                because no getnum() function is available
                under C/400, ANSI C or  SAA C.
                Note that an element in a structure is referred
                to its qualified name frecord.camt*/

          printf("Enter $ to end :\n");
          c=getchar();

          if (c!='$')
           {
```

```
                fwrite(&frecord,sizeof(frecord),1,file1);
                    /* write to file from record buffer. Note
                       that the address (pointer) is used */
                }
            }
        fclose(file1);
}

readfld(buffer,size) /* read field with getchar.
                        The reason for defining readfld function
                        instead of using scanf or gets function
                        is because these functions will consider
                        a space as a terminator. If the program
                        consecutively contains 3 scanf statements
                        for 3 fields and the user enters
                        "David Choi & Co." from the screen, the
                        program interprets as 4 fields have been
                        entered ("David","Choi","&","Co.") and
                        only the first scanf will be executed
                     */

char *buffer;
int size;

{
   int i;
   char c;

   i=0;
   while((c=getchar())!='\n')  /* read character until
                                  enter key is pressed */
   {
      *buffer++ = c;
      i=i+1;
      }

   *buffer&supplus,*='\0';  /* A string must end with NULL character
                       (Hex 00), therefore a NULL character is
                       added as last character of the string,
                       otherwise when C try to read the field
                       again, it will not know where the string
                       is terminated.
                    */

   }
```

A.5 Example 4 - PGM04 (Reading Records from a File by C)

This program reads the records created by the program in Example 3. Each record is displayed on the screen after a read until the end of the file is reached.

```
/* ------------------------------------------------

   Program Name : PGM04

   This program read and display the records written by
   program in Example 3.

   ------------------------------------------------ */

#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
 FILE *file1;
 char c;

struct frecord1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          int camt;
          } frecord;

    file1=fopen("TFILEB","rb type=record");

    while(!feof(file1))    /* feof is used to test for
                              end of file condition     */
     {
        fread(&frecord,sizeof(frecord),1,file1);

        if(!feof(file1))
        {

        printf("Number = %-5s\n",frecord.cnum);
        printf("Name = %-30s\n",frecord.cname);
        printf("Address = %-30s\n",frecord.cadd);
        printf("Phone = %-10s\n",frecord.ctel);
        printf("Amount = %d\n",frecord.camt);
        printf("??/n");
        c=getchar();
        }
      }
    fclose(file1);
}
```

A.6 Example 5 - PGM05 (Read Record from AS/400 Database File)

This program reads records in an AS/400 Database FRPG and displays them. QXXFORMAT routine is used here as an illustration. It is important to note that QXXFORMAT is applicable to AS/400 only. Therefore this code is not portable to other system.

It is also important to note that since C/400 does not have data type packed or zoned, if the database file contain any such field, they should first be defined as alphanumeric field and then later use QXX.... data conversion routine to convert them to C/400 numeric data.

Layout of the AS/400 database file FRPG is as follows:

```
    Format name : LABF
    Fields :  CNUM     length = 5    type = alphanumeric
              CNAME    length = 30   type = alphanumeric
              CADD     length = 30   type = alphanumeric
              CTEL     length = 10   type = alphanumeric
```

IMPORTANT: This example also illustrates the difference between a C/400 character string and a AS/400 string. In C/400, all strings are ended with NULL character (HEX 00). In AS/400, the NULL character may be just one of the possible characters in a string. When running this program, all the field alignments are correct, but the strings are terminated incorrectly because no NULL characters were detected. Therefore, when CNUM is displayed, it starts from the first position of CNUM, then all the way to the end of the record. For CNAME, it starts from first position of CNAME (which is the 6th position from the beginning of the record) and so on.

Therefore, in order to process a string read from an AS/400 database file (especially when record of the file is created by other languages such as COBOL and RPG), the C program should add a NULL character to end of it in order to display the fields correctly.

```
/* ------------------------------------------------

  Program Name : PGM05

  This program reads records from an AS/400 database file
  FRPG sequentially and displays the record on screen.

  QXXFORMAT is used to specify the record format this
  C/400 program will work on. However, QXXFORMAT is an
  C/400 extension and can be used only on AS/400.

  In this example, since the physical database contains
  only one record format, usage of QXXFORMAT is optional.
  That is, the same result can be achieved by using a standard
  fread.

  ------------------------------------------------  */

#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
```

```
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
 FILE *file1;
 char c;

struct frecord1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          } frecord;     /* This structure is the same as the
                            layout defined in database file FRPG.
                            Note that only alphanumeric fields are
                            used in this example.
                            If there are numeric fields (packed or
                            zoned) they should be defined as
                            alphanumeric fields in this structure.
                            After the field is read into this
                            structure, the program should then use
                            QXX.... conversion routines from
                            EPM application library to do the
                            data conversion before the C program
                            can use it for numeric processing.
                         */

    file1=fopen("FRPG","rb type=record");

    QXXFORMAT(file1,"LABF      ");
              /* QXXFORMAT is a File Routine to specify
                 record format to the C/400 program.
                 This is a C/400 extension. It is not
                 available in ANSI C or SAA C.
              */

    while(!feof(file1))
     {
        fread(&frecord,sizeof(frecord),1,file1);
        if(!feof(file1))
         {
        printf("Number = %5s\n",frecord.cnum);
        printf("Name = %30s\n",frecord.cname);
        printf("Address = %30s\n",frecord.cadd);
        printf("Phone = %10s\n",frecord.ctel);
        printf("\n");
        c=getchar(); /* After the user pressed enter key
                        the program will continue        */
         }
      }

    fclose(file1);
}
```

A.7 Example 6 - PGM06 (Working with Database and Display File)

This is an example of how C/400 reads a record from AS/400 database file and writes it to a display file.

Layout of display file SRPGS is as follows:

```
     A*  89/04/25  15:22:29    IBM         REL-R01M02  5728-PW1
     A*            16:11:05    DAVID       REL-R06M00  5714-UT1
     A                                      DSPSIZ(24 80 *DS3)
     A                                      PRINT
     A          R SCRF
     A*  89/04/25  15:22:29    IBM         REL-R01M02  5728-PW1
     A*            16:11:05    DAVID       REL-R06M00  5714-UT1
     A                                      BLINK
     A                                    3 19'Demonstration Screen for Inquiry'
     A                                        DSPATR(HI)
     A                                        DSPATR(UL)
     A                                    5 56'By David Choi'
     A                                    9 11'Customer Number :'
     A            CNUM           5A  B  9 32DSPATR(HI)
     A                                        DSPATR(CS)
     A                                        DSPATR(UL)
     A                                        DSPATR(PC)
     A                                   11 22'Name :'
     A            CNAME         30A  O 11 32DSPATR(HI)
     A                                   13 19'Address :'
     A            CADD          30A  O 13 32DSPATR(HI)
     A                                   15 17'Telephone :'
     A            CTEL          10A  O 15 32DSPATR(HI)
     A            COPT           1A  B 19 42DSPATR(HI)
     A                                        DSPATR(CS)
     A                                        DSPATR(UL)
     A                                   19 13'Enter <E> to End =>'
```

```
/* ------------------------------------------------------------

   Program Name : PGM06

   This program reads a record from database file sequentially.
   The contents of the record is then moved to a display file and then
   displayed on the screen.

  ------------------------------------------------------------ */

#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
 FILE *file1;
 FILE *file2;
 char c;
```

```
struct frecord1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          } frecord;  /* structure for database file buffer */

struct fscrout1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          char copt;
          } fscrout;  /* structure for screen output buffer */

struct fscrin1 {
          char cnum[5];
          char copt;
          } fscrin;   /* structure for screen input buffer */

    file1=fopen("FRPG","rb type=record");

    file2=fopen("SRPGS","rb+ type=record");
                  /* SRPGS is a display file */

    QXXFORMAT(file1,"LABF      "); /* database record will be
                                       read                    */

    while(!feof(file1))
     {
        fread(&frecord,sizeof(frecord),1,file1);
                 /* read one database record        */

        if(!feof(file1))
        {
        clrout(&fscrout,sizeof(fscrout));
        clrout(&fscrin,sizeof(fscrin));
               /* both input and output screen buffer are
                  initialized with spaces. This is to ensure
                  not data from the last screen write will
                  remain in the output buffer              */

        strcpy(fscrout.cnum,frecord.cnum);
        strcpy(fscrout.cname,frecord.cname);
        strcpy(fscrout.cadd,frecord.cadd);
        strcpy(fscrout.ctel,frecord.ctel);
               /* content of database records are copied into
                  the output screen buffer using strcpy       */

        fscrout.copt = ' ';

        QXXFORMAT(file2,"SCRF      ");
               /* prepare to write to display file with
                  format SCRF                              */

        fwrite(&fscrout,sizeof(fscrout),1,file2);
               /* write to display file                    */

        fread(&fscrin,sizeof(fscrin),1,file2);
```

```
                    /* read the display file                    */
            }
          }
        fclose(file1);
        fclose(file2);
}

clrout(buffer,j) /*  fill output buffer with space  */
char *buffer;
int j;

{
   int i;
   for(i=0;i<j;i++)
     {
        *buffer++ = ' ';
        }
  }
```

A.8 Example 7 - PGM07 (Working with Printer File)

This program writes heading line to printer file, read database record and print each detail line to printer file. At the end ending line will be printed.

DDS of the printer file SAM17P is as follows:

```
     A*
     A          R HEADLINE1                 TEXT('FIRST HEADING LINE')
     A                                      SKIPB(4)
     A                                     30'CUSTOMER ORDER REPORT'
     A                                    +10'DATE :'
     A                                     +1DATE
     A                                       EDTCDE(Y)
     A                                    +10'PAGE :'
     A                                     +1PAGNBR
     A                                       SPACEA(3)
     A*
     A                                      1'CUST NO.'
     A                                     11'NAME'
     A                                     41'ADDRESS'
     A                                     71'ORDER NUM'
     A                                     81'ORDER DESCRIPTION'
     A                                    101'AMOUNT'
     A                                       SPACEA(1)
     A                                      1'------------------------------'
     A                                     31'------------------------------'
     A                                     61'------------------------------'
     A                                     91'----------------------'
     A                                       SPACEA(3)
     A*
     A          R DETAIL1                   TEXT('FORMAT FOR DETAIL LINE')
     A            CNUMBER        5   0      1
     A            CNAME         30   0     11
     A            CADDRESS      30   0     41
     A            CTELEPHONE    10   0     71
     A                                       SPACEA(1)
     A*
     A*
     A*
     A          R ENDING1                   TEXT('FORMAT FOR ENDING')
     A                                      SPACEB(2)
     A                                     40'******************'
     A                                     +2'END OF REPORT'
     A                                     +2'******************'
     A                                       SKIPA(1)
/* ---------------------------------------------------

   Program Name : PGM07

   This program writes a heading line to the printer file, then
   starts reading the database record, prints a detail line to the printer
   file. At the end of database file read, an ending line will be
   printed.

--------------------------------------------------- */
```

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
 FILE *file1;
 FILE *file2;
 char c;

struct frecord1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          } frecord;    /* structure for database record buffer */

struct fscrout1 {
          char cnum[5];
          char cname[30];
          char cadd[30];
          char ctel[10];
          } fscrout;    /* structure for printer output detail
                           line buffer                          */

struct fdummy {
          char a1;
          } fdummy;    /* define a dummy output buffer field */

    file1=fopen("LABMASTER","rb type=record");

    file2=fopen("SAM17P","wb type=record");   /* SAM17P is printer file */

    QXXFORMAT(file2,"HEADLINE1 ");
    fwrite(&fdummy,sizeof(fdummy),1,file2);
         /* these two lines write heading to printer file.
            Another way to code the statement is :
            fwrite("",0,0,file2); meaning not output data
            send, but write the heading.
            Same apply to ending line.*/

    QXXFORMAT(file1,"LABMASTERF");
         /* prepare to read database file   */

    while(!feof(file1))
     {
        fread(&frecord,sizeof(frecord),1,file1);
              /* read database file record */

        if(!feof(file1))
        {
        clrout(&fscrout,sizeof(fscrout));
             /* output buffer is cleared first to ensure no
                data from previous write                    */
```

```
        strncpy(fscrout.cnum,frecord.cnum,sizeof(frecord.cnum));
        strncpy(fscrout.cname,frecord.cname,sizeof(frecord.cname));
        strncpy(fscrout.cadd,frecord.cadd,sizeof(frecord.cadd));
        strncpy(fscrout.ctel,frecord.ctel,sizeof(frecord.ctel));
                /* content of database record is moved into
                   output buffer                              */

        QXXFORMAT(file2,"DETAIL1   ");
        fwrite(&fscrout,sizeof(fscrout),1,file2);
                /* write detail line from buffer to printer
                   file                                       */

        }
      }

    QXXFORMAT(file2,"EDNING1   ");
    fwrite(&fdummy,sizeof(fdummy),1,file2);
            /*  write ending line to printer file     */

    fclose(file1);
    fclose(file2);
}

clrout(buffer,j) /*  fill output buffer with space  */
char *buffer;
int j;

{
   int i;
   for(i=0;i<j;i++)
     {
        *buffer++ = ' ';
        }
  }
```

A.9 Example 8 - PGM08 (Working with Display File with INDARA)

This program illustrate how to use INDARA to control indicators in a display file. By defining INDARA in a display file, a separate buffer of 99 bytes is reserved for the indicators. In the program, an area of 99 bytes is also defined to interface with the indicator buffer defined in the display file.

Display files with indicators are unique in AS/400 and the functions are not portable to another system under SAA.

Layout of display file used in program is as follow:

```
     A*  89/06/19  12:45:07    LAMONT      REL-R02M00  5728-PW1
     A                                      DSPSIZ(24 80 *DS3)
     A                                      INDARA
     A                                      CA03(03 'END OF JOB')
     A          R FORMAT1
     A*  89/06/19  12:45:07    LAMONT      REL-R02M00  5728-PW1
     A                                      BLINK
     A                                   4 20'Testing Screen with Indicators'
     A                                      DSPATR(HI)
     A                                      DSPATR(UL)
     A                                  11 11'Enter your option or <E> to End =>'
     A            COPTION        1   B 11 49DSPATR(HI)
     A            ERRORMSG      50   O 15 15DSPATR(HI)
     A                                      DSPATR(BL)
     A                                  17 12'F-3 = End of Job'
```

```
*****************************************************************

/* ------------------------------------------------------------

  Program Name : PGM08

  This program illustrates how to work with a display file using
  an indicator area (INDARA).

  Note that a separate indicator area for 99 indicators is
  defined in the program. The display file automatically passes the
  indicators to this area and vice versa.

  The routine QXXSINDARA is used to define the indicator area
  that will be used together with the display file.

------------------------------------------------------------ */
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
 union     /* The union defines that two structures will share
              the same memory space                              */
  {
   struct
```

```
   {
    char in01;
    char in02;
    char in03;
    char in04[96];
    } str;         /* structure str contains 99 bytes representing
                      99 indicators as a result of defining
                      INDARA in display file.*/

  char array[99];
   } indic;         /* india is an array of 99 bytes. This is for the
                       convenience of initializing the indicator area */

struct fscrout1 {
          char coption;
          char errormsg[50];
          } fscrout;  /* screen output buffer   */

struct fscrin1 {
          char coption;
          } fscrin;   /* screen input buffer   */

FILE *file1,

file1=fopen("S16SCR1","ab+ type=record indicators=y");
                  /* indicators=y indicates both the C
                     program and display file will
                     assign a separate area to store
                     indicator value (99 altogether).
                     Therefore no space defined for
                     indicator appears in both screen
                     input and output buffer. */

QXXSINDARA(file1,indic.array); /* prepare to work on indicators */

indic.str.in03 = 0XF0;
                      /* indicator value can either be HEX 00
                         or HEX 01. Here, indicator 3 is
                         initialized to HEX 00 by 0XF0. */

clrout(&fscrout,sizeof(fscrout)); /* fill output buffer with
                                     space                  */

QXXFORMAT(file1,"FORMAT1   ");    /* will work on screen with
                                     format FORMAT1.          */

fwrite(&fscrout,sizeof(fscrout),1,file1);  /* write screen */

fread(&fscrin,sizeof(fscrin),1,file1);     /* read screen  */

fscrout.coption = fscrin.coption;   /* move data from input
                                       buffer to output buffer */

if(indic.str.in03 == 0XF1)   /* if operator does not press F-3 */
 {
  strcpy(fscrout.errormsg,"You have pressed F-3");
                           /* move this message to output */
  }
  else
```

```
  {
  strcpy(fscrout.errormsg,"You did not press F-3");
                            /* otherwise move this message */
  }

  setmsg(&fscrout.errormsg,sizeof(fscrout.errormsg));
               /* setmsg is called to fill empty space with space.
                  In C, empty space left are padded with HEX 00
                  and cannot be displayed on screen. This function
                  is to replace all HEX 00 with space. */

fwrite(&fscrout,sizeof(fscrout),1,file1);  /* write screen */

fread(&fscrin,sizeof(fscrin),1,file1);     /* read screen  */
}

clrout(buffer,j) /*  fill output buffer with space  */
char *buffer;
int j;

{
   int i;
   for(i=0;i<j;i++)
     {
        *buffer++ = ' ';
        }
  }
setmsg(buffer,j) /*  fill output buffer with space  */
char *buffer;
int j;

{
   int i;
   for(i=0;i<j;i++)
     {
        if(*buffer=='\0')   /* replace all HEX 00 with space */
        {
        *buffer = ' ';
        }
        *buffer++;
        }
  }
```

A.10 Example 9 - PGM09 (Writing Records to a Database File with C)

This program reads data from the screen and writes the record to a database file.

This is a good example to highlight the difference between a field generated by C and fields generated by other languages such as COBOL and RPG.

A C string always ends with a null character. When a string is read from the screen into the record buffer, all spaces after the last character entered from screen the will be null characters. If this string is directly written to the record, it cannot be used in the display. So before writing to record, all NULL characters are replaced with spaces. If the string's NULL characters are not replaced by spaces, then this field can only be displayed by C program since only C knows how to display it.

Another comment on this program is about the numeric field. The database has defined a zoned field. Since C/400 (also in SAA C and ANZI C) does not provide getnum() function, the number can only be read as a string and data in the string is left-justified. In order to make this field compatible with an AS/400 zoned decimal format, the user can use QXXITOZ data conversion routine as described in the *C/400 User's Guide*, or use the method described in this example. But regardless of which method you use, the converted number will always be written to the record as a character field, since C/400 does not have a data type zoned. The same is true for packed decimal.

Layout of Database file FSAM19 used in program is:

```
     A          R FORMAT1
     A            FIELD1         5          COLHDG('FIELD 1')
     A            FIELD2        30          COLHDG('FIELD 2')
     A            FIELD3         8S 0       COLHDG('FIELD 3')
```

```
/* ------------------------------------------------------------

   Program Name : PGM09

   This program reads data from the screen, reformats the data by
   replacing all NULL characters (HEX 00) with spaces in a
   string, fills with leading zero and right-justifies the numeric
   string.

------------------------------------------------------------ */

#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <xxasio.h>
#include <xxfdbk.h>
#include <stdlib.h>

main()
{
 struct frecord
 {
  char cnum[5];
  char cname[30];
```

```
 char camt[8];
 }
  frecord;   /* structure for record buffer    */

char x;
char xamt[8];

FILE *file1;

file1=fopen("FSAM19","wb+ type=record");

x=' ';

while(tolower(x)!='e')
{
 /*   ---- clear buffer first -------- */

 clrbuf(&frecord,sizeof(frecord));

 printf("Enter cnum :\n");  /* prompting message */
 readstr(&frecord.cnum,sizeof(frecord.cnum));
          /* read string into frecord.cnum by calling
             function readstr                          */

 printf("Enter cname :\n");
 readstr(&frecord.cname,sizeof(frecord.cname));

 printf("Enter camt :\n");
 readstr(frecord.camt,sizeof(frecord.camt));

 printf("Enter <e> to end :\n");
 x=getchar();

 if (tolower(x)!='e')
   {
    pad(&frecord.cnum,sizeof(frecord.cnum));
         /* fill all NULL characters in string with spaces */

    pad(&frecord.cname,sizeof(frecord.cname));

    padnum(&frecord.camt,sizeof(frecord.camt));
        /*  This is a manual method of converting an
            integer read from screen into a zoned decimal
            that can be written to file.
            frecord.camt is first read from screen, but
            at this time the number in this character string
            is left-justified.
            This string is then converted to a string filled
            with leading zeros and a right-justified string
            by calling padnum.

            Another way of doing the same function is through
            QXXITOZ. The string will first be read from
            screen, then it will be converted to an integer
            by atoi() function. The integer will then be
            converted to zone decimal by call the QXXITOZ
            function.
        */
```

```
     fwrite(&frecord,sizeof(frecord),1,file1);

     }   /* --- end of if  */

    } /* ---- end of while */

 } /* ---- end of main */

pad(buffer,l)   /* replace all NULL characters with spaces */

char *buffer;
int l;

{
 int i;

 for(i=0;i<l;i++)
  {
   if(buffer[i]=='\0')
    buffer[i]=' ';
    }
}

padnum(buffer,l)   /* re-arrange digits in string with leading
                      zero and make it right-justified        */

char *buffer;
int l;

{
 int i;
 int j;
 int k;
 char s[8];

 i=atoi(buffer);

 sprintf(s,"%d",i);


  k=0;
  for(j=0;j<sizeof(s);j++)
  {
   if(!isdigit(s[j]))   /* if digit is not a printable character */
    {
     k=k+1;
     }
   }
   for(j=0;j<k;j++)
   {
    buffer[j]='0';
    }
   for(j=k;j<sizeof(s);j++)
    {
     buffer[j]=s[j-k];
     }
}

readstr(buffer,l)  /*  read data from screen   */
```

```
char *buffer;
int l;
 {
 char c;

  while((c=getchar())!='\n')
   {
   *buffer=c;
   *buffer++;
    }
 }

clrbuf(buffer,l)    /* clear buffer with spaces   */

char *buffer;
int l;

{
 int i;

 for(i=0;i<l;i++)
 {
  buffer[i]=' ';
  }
 }
```

A.11 Example 10 - PGM10 (Dynamic SQL under Commitment Control)

See the section on Dynamic SQL in Chapter 4 for a discussion of this sample program. The following is the CL-Program as well as the C program and the display file.

```
Display file used in example 10
     A                                      DSPSIZ(24 80 *DS3)
     A                                      MSGLOC(24)
     A                                      PRINT
     A                                      CA03(03 'Exit')
     A                                      CA08(08 'Prepare Update')
     A                                      CA09(09 'Prepare Delete')
     A                                      CA10(10 'Execute')
     A                                      CA11(11 'Rollback')
     A                                      CA12(12 'Commit')
     A          R FMT1
     A                                      SETOF(88)
     A                                      OVERLAY
     A  61N63                            9  2'UPDATE'
     A                                      COLOR(WHT)
     A  62N63                            9 16'Table    '
     A                                      DSPATR(RI)
     A  61N63                            9 19'SET'
     A                                      COLOR(WHT)
     A  61N63                            9 24'Column  '
     A                                      DSPATR(RI)
     A  61N63                            9 33'='
     A                                      COLOR(WHT)
     A  61N63                            9 37' ? '
     A                                      DSPATR(RI)
     A  62N63
     AO 61N63                            9 44'WHERE'
     A                                      COLOR(WHT)
     A  62N63
     AO 61N63                            9 50'Column  '
     A                                      DSPATR(RI)
     A  62N63
     AO 61N63                            9 63'Op'
     A                                      DSPATR(RI)
     A  62N63
     AO 61N63                            9 72' ? '
     A                                      DSPATR(RI)
     A  61N63                           10  9'MATABLE'
     A  61N63                           17  9'KITABLE'
     A  61N63                           10 24'VNAME'
     A  61N63                           11 24'NAME'
     A  61N63                           12 24'STAATA'
     A  61N63                           13 24'PLZ'
     A  61N63                           14 24'ORT'
     A  61N63                           15 24'STR'
     A  61N63                           17 24'KNAME'
     A  61N63                           18 24'KKIGEL'
     A                                   1 20'Dynamic SQL in an C/400 Program'
     A                                      COLOR(WHT)
     A  61N63     TABLEU         8A  I  8  9DSPATR(PC)
     A                                      VALUES('MATABLE ' 'KITABLE')
     A                                      CHANGE(50)
     A  62N63     TABLED         8A  I  8 16DSPATR(PC)
```

```
A                                       VALUES('MATABLE ' 'KITABLE')
A                                       CHANGE(50)
A  61N63     COLMNU         6A  I  8 24VALUES('VNAME ' 'NAME  ' 'STAATA' '-
A                                       PLZ   ' 'ORT   ' 'STR   ' 'KNAME ' -
A                                       'KKIGEL')
A                                       CHANGE(51)
A  62N63
AO 61N63     COLMNW         6A  I  8 50VALUES('VNAME ' 'NAME  ' 'STAATA' '-
A                                       PLZ   ' 'ORT   ' 'STR   ' 'KNAME ' -
A                                       'KKIGEL')
A                                       CHANGE(52)
A  62N63
AO 61N63     OPC            2A  I  8 63VALUES(' =' '= ' ' >' '> ' ' <' '< -
A                                       ' '>=' '<=')
A                                       CHANGE(53)
A  62N63
AO 61N63                              10 63' ='
A  62N63
AO 61N63                              11 63' >'
A  62N63
AO 61N63                              12 63' <'
A  62N63
AO 61N63                              13 63'>='
A  62N63
AO 61N63                              14 63'<='
A  61 63     FRM           20A  I 15 27CHECK(LC)
A  61 63                              16 41'|'
A                                       COLOR(RED)
A  61 63                              17 41'|'
A                                       COLOR(RED)
A  61 63                              18 41'|'
A                                       COLOR(RED)
A  61 63                              19 41'v'
A                                       COLOR(RED)
A  61 63
AO 62 63     WHE           20A  I 15 51CHECK(LC)
A  61 63
AO 62 63                              16 60'|'
A                                       COLOR(RED)
A  61 63
AO 62 63                              17 60'|'
A                                       COLOR(RED)
A  61 63
AO 62 63                              18 60'|'
A                                       COLOR(RED)
A  61 63
AO 62 63                              19 60'v'
A                                       COLOR(RED)
A            S             70A  O 21  6COLOR(BLU)
A                                     20  6'Status of the prepared SQL stateme-
A                                       nt'
A                                       COLOR(BLU)
A                                       DSPATR(RI)
A  88                                 20 61'Error-Code:'
A                                       COLOR(WHT)
A  88        SQLDEC         5Y 00 20 74EDTCDE(M)
A                                       COLOR(WHT)
A  62N63
AO 61N63                              10 50'VNAME'
```

```
     A  62N63
     AO 61N63                        11 50'NAME'
     A  62N63
     AO 61N63                        12 50'STAATA'
     A  62N63
     AO 61N63                        13 50'PLZ'
     A  62N63
     AO 61N63                        14 50'ORT'
     A  62N63
     AO 61N63                        15 50'STR'
     A  62N63
     AO 61N63                        17 50'KNAME'
     A  62N63
     AO 61N63                        18 50'KKIGEL'
     A                               23  2'Function keys:'
     A                               23 17'3=Exit  8=Prepare Update  9=Prepar-
     A                                    e Delete  10=Execute'
     A                               24 17'11=Rollback  12=Commit'
     A  61N63                         9  9'Table    '
     A                                    DSPATR(RI)
     A  62N63                         9 10'FROM'
     A                                    COLOR(WHT)
     A  62N63                         9  2'DELETE'
     A                                    COLOR(WHT)
     A  62N63                        10 16'MATABLE'
     A  62N63                        17 16'KITABLE'
     A  61N63
     AO 62N63                         4  2'                          Prepare  --
     A                                     Mode                              -
     A                                           '
     A                                    DSPATR(RI)
     A  63                            4  2'                          Execute  --
     A                                     Mode                              -
     A                                           '
     A                                    DSPATR(RI`
```

CL program for Start and End Commitment Control

```
            PGM
            STRCMTCTL  LCKLVL(*CHG)
            CALL       PGM(DYSQL)
            ENDCMTCTL
            ENDPGM
```

C/400 programm using dynamic SQL programming technique

```
/*****************************************************************/
/*                                                               */
/* PROGRAM I.D.   :  example 10                                  */
/* AUTHOR         :                                              */
/* DATE           :  june    1989                                */
/*                                                               */
/*****************************************************************/

#include <stdlib.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>

   /***************************/
   /* Global Data declarations */
```

```
   /***************************/

   FILE *dspf1;         /* Pointer to locate display file*/
/**  SQL Communication Area    ************************************/
     EXEC SQL INCLUDE SQLCA;
/****DSPF Input buffer ********************************************/
      struct ibild
       { char in03;     /* exit key */
         char in08;      /* prepare update */
         char in09;      /* prepare delete */
         char in10;      /* execute  */
         char in11;      /* rollback */
         char in12;      /* commit   */
         char in88;      /* commit   */
         char in50;
         char in51;
         char in52;
         char in53;
         char tableu [8];
         char tabled [8];
         char colmnu [6];
         char colmnw [6];
         char opc    [2];
         char frm    [20];
         char whe    [20];
     } ibild;

/**  Display file output buffer**************************************/
   struct obild {
          char in61;
          char in63;
          char in62;
          char in88;
          char s        [70];
        double sqldec;
          } obild;

/**  UPDATE Statement that the PREPARE is naming STMT*************** /
   struct xupd
   {
   char upd [7];
   char tbl [8];
   char set [7];
   char col1 [6];
   char eql  [8];
   char usg1 [2];
   char wher [6];
   char col2 [6];
   char opc  [2];
   char usg2 [2];
   };

/**  DELETE Statement that the PREPARE statement is naming STMT ******/
   struct xdel
   {
   char del [12];
   char tbl [8];
   char wher [24];
   char col1 [6];
```

```
    char opc  [2];
    char usg1 [2];
    };

 /**  Redefine the structures, because a character string is needed ***/
 /**  for the PREPARE statement    ***********************************/
    union uupd {
    struct xupd supd;
    char updstr [56];} updat;
    union udel {
    struct xdel sdel;
    char delstr [56];} delet;
    /*******************************/
    /* End Globla Data Declaration */
    /*******************************/

 main()
 {
    /*******************************/
    /* Function Prototyping        */
    /*******************************/
void prepupd();
void prepdel();
void execstmt();
int clrout();


   /*******************************/
   /* open display file           */
   /*******************************/

    dspf1= fopen ("SQL2D", "ab+ type=record");

   /*****************************************/
   /* set indicators, fill buffer with blank */
   /*****************************************/
        clrout(&obild,sizeof(obild));
        clrout(&ibild,sizeof(ibild));

        obild.in61 = 0XF1;
        ibild.in03 = 0XF0;

        strcpy(updat.updstr,
         "UPDATE 12345678   SET 123456      = ? WHERE 12345612 ?");
        strcpy(delet.delstr,
         "DELETE FROM 12345678                 WHERE 12345612 ?");
        strcpy(obild.s,updat.updstr);

        /*************************************/
        /* carry on if F3 is not hit by user */
        /*************************************/
    while ( ibild.in03 != 0XF1)
      {
        /*************************************************/
        /* Display sql2d                                 */
        /*************************************************/
    QXXFORMAT(dspf1,"FMT1      ");
    fwrite(&obild,sizeof(obild),1,dspf1);
    fread(&ibild,sizeof(ibild),1,dspf1);
```

```
if (ibild.in03 != 0XF1 )
   {
    if (ibild.in08 == 0XF1 )
       {
        obild.in61 = 0XF0;
        obild.in62 = 0XF0;
        obild.in63 = 0XF0;
        obild.in61 = 0XF1;
        strcpy(obild.s,updat.updstr);
       }                                          /*  in08 == 1 */
    else                                          /*  in08 != 1 */
       if (ibild.in09 == 0XF1 )
          {
           obild.in61 = 0XF0;
           obild.in62 = 0XF0;
           obild.in63 = 0XF0;
           obild.in62 = 0XF1;
           strcpy(obild.s,delet.delstr);
          }                                       /*  in09 == 1 */
       else                                       /*  in09 != 1 */
          if (ibild.in10 == 0XF1 )
             {
              EXEC SQL
              PREPARE STMT FROM :obild.s;         /*   PREPARE  */
              if (sqlca.sqlcode == 0)             /* SQL Fehler? */
                {
                 obild.in63 = 0XF1;
                }   /* sqlcode == 0 */
              else
                {
                 obild.in88 = 0XF1;
                /*  obild.sqldec = sqlca.sqlcode; */
printf("SQLCODE ist     %d??/n ",sqlca.sqlcode);
                }                                 /* end SQL Fehler */
             }   /* in10 == 1 */
          else
             {
              if (ibild.in11  == 0XF1)
                {
                 EXEC SQL
                 ROLLBACK;                        /*  ROLLBACK  */
                 obild.in63  == 0XF0;
                }                                 /* in11 == 1 */
              else                                /* in11 != 1 */
                {
                 if (ibild.in12 == 0XF1)
                   {
                    EXEC SQL
                    COMMIT;                       /*  COMMIT  */
                    obild.in63  == 0XF0;
                   }                              /* in12 == 1 */
                 else                             /* in12 != 1 */
                   {
                    if (obild.in63 == 0XF1)
                      {
                            execstmt();
                      }                           /* in63 == 1 */
                    else                          /* in63 != 1 */
                    {
```

```
                        if (obild.in61 == 0XF1)
                          {
                               prepupd();
                          }                    /* in61 == 1 */
                        else                   /* in61 != 1 */
                        {
                           if (obild.in62 == 0XF1)
                             {
                                  prepdel();
                             }                 /* in62 == 1 */
                        }                      /* in61 != 1 */
                      }                        /* in63 != 1 */
                    }                          /* in12 != 1 */
                  }                            /* in11 == 1 */
                }                              /* in10 != 1 */
              }                                /* in03 != 1 */
    }                                          /*  while */
    fclose(dspf1);
}                                               /*  main  */

 /*******  Execute Statement    ***********************************/
void execstmt()
{
   if (obild.in61 == 0XF1)
     {
     EXEC SQL
     EXECUTE STMT USING :ibild.frm, :ibild.whe;
                 if (sqlca.sqlcode != 0)
                   {
                    obild.in88 = 0XF1;
               /*   obild.sqldec = sqlca.sqlcode;   */
    printf("SQLCODE ist     %d??/n ",sqlca.sqlcode);
                   }  /* sqlcode != 0 */
     }
   else
     {
      EXEC SQL
      EXECUTE STMT USING :ibild.whe;
      if (sqlca.sqlcode != 0)
        {
         obild.in88 = 0XF1;
/*       obild.sqldec=  sqlca.sqlcode;  */
      printf("SQLCODE ist     %d??/n ",sqlca.sqlcode);
        }  /* sqlcode != 0 */
     }
 }    /* execute statement function */

/*****************************************************************/
/*******  Prepare Update       ***********************************/
void prepupd()
{
    strcpy(updat.supd.tbl,ibild.tableu);
    strcpy(updat.supd.set,"   SET ");
    strcpy(updat.supd.col1,ibild.colmnu);
    strcpy(updat.supd.eql,"      = ");
    strcpy(updat.supd.usg1,"? ");
    strcpy(updat.supd.wher,"WHERE ");
    strcpy(updat.supd.col2,ibild.colmnw);
    strcpy(updat.supd.opc,ibild.opc);
```

```
    strcpy(updat.supd.usg2,"? ");
    strcpy(obild.s,updat.updstr);
}    /* end function Prepare Update   */

/*****************************************************************/
/*******  Prepare Delete       *********************************/
void prepdel()
{
    strcpy(delet.sdel.del,"DELETE FROM ");
    strcpy(delet.sdel.tbl,ibild.tabled);
    strcpy(delet.sdel.wher,"                  WHERE ");
    strcpy(delet.sdel.col1,ibild.colmnw);
    strcpy(delet.sdel.opc,ibild.opc);
    strcpy(delet.sdel.usg1,"? ");
    strcpy(obild.s,delet.delstr);
}    /* end function Prepare Delete   */
/*****************************************************************/
/******  Clears input and output buffer    **********************/
clrout(buffer,j) /*  fill output buffer with space  */
char *buffer;
int j;

{
  int i;
  for(i=0;i<j;i++)
    {
       *buffer++ = ' ';
       }

}    /* end function clrout    */
/*********** End of Example 10  ****************************/
```

Index

A
Abstract iii
Alignment of Data 25
Application Performance Tuning Aid 23

B
Bibliography ix
Binary File 33
Binary Stream 30
Buffered Data Stream 30
Buffered Input 7

C
Case Sensitivity 7
Character Set Required 5
Character String 33
Commit 44
Commitment control 21
Conversion Routines 21
CRTBNDPGM 23
CRTCPGM 16
CRTSQLC 17, 46

D
Data Mapping 41
Debug 16
Display File 57
Dynamic SQL 50

E
EPM 11
EPM Application Library 20
EPM Environment 13
Exception Handler 17
Extended Program Model 13
External Variables 12
Externally Described File 27
EXTPGMINF 22

F
Floating Point 8, 48

G
Graphics 5

H
History of C 1
Host Variables 46

I
IEEE-488 8
Indicators 57

L
Level Checks 2
Levels of C Language 3
Location of C/400 Runtime Routines 24
Logical File 27, 42

M
Multiple Entry Points 12

O
Observability 19
OPNQRYF 43

P
PAG 14
PC Emulation 5
Performance 63
Pointer Usage 7
Portability of C 3
Preface. vii
Print Key 61
Publications ix
PURGE 14

R
Record length 25, 33
Return Codes 8

S
SAA 1, 3, 46
Session Manager 19
SETPGMINF 14, 16, 20, 22
Signal Handler 17
SQL 36, 46
SSNATTR 20
Static SQL 49
Stream Mode 30
Strength of C Language 2
Structure 25

T

Table Of Contents xiii
Text File 33
Text Stream 30
Trigraphs 6

READER'S COMMENTS

Title: **AS/400 C Language Introduction**

Document Number: **GG24-3434-00**

You may use this form to communicate your comments about this publication, its organization or subject matter with the understanding that IBM amy use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Comments:

Reply Requested: ____ Yes ____ No

Name: _____

Job Title: _____

Address: _____

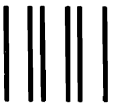
_____

_____

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 40 ARMONK, N Y

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Technical Support Center
Department 977, Building 003-1
3605 Highway 52N
Rochester, Minnesota 55901

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

READER'S COMMENTS

Title: **AS/400 C Language Introduction**

Document Number: **GG24-3434-00**

You may use this form to communicate your comments about this publication, its organization or subject matter with the understanding that IBM amy use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Comments:

Reply Requested: ____ Yes ____ No

Name: _____

Job Title: _____

Address: _____

_____

_____

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 40 ARMONK, N Y

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Technical Support Center
Department 977, Building 003-1
3605 Highway 52N
Rochester, Minnesota 55901

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

GG24-3434-00

AS/400 C Language Introduction

GG24-3434-00

PRINTED IN THE U.S.A.