SAS/C® CICS User's Guide, Release 7.00

The Power to Know™

SAS Publishing

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/C® CICS User's Guide, Release 7.00*, Cary, NC: SAS Institute Inc., 2001.

SAS/C® CICS User's Guide, Release 7.00

Copyright © 2001 by SAS Institute Inc., Cary, NC, USA.

1–58025–728–3

All rights reserved. Produced in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

U.S. Government Restricted Rights Notice. Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227–19 Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, April 2001

SAS Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, CD-ROM, hard copy books, and Web-based training, visit the SAS Publishing Web site at www.sas.com/pubs or call 1-800-727–3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

IBM® and all other International Business Machines Corporation product or service names are registered trademarks or trademarks of International Business Machines Corporation in the USA and other countries.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

Chapter 1 △ Overview of the SAS/C and CICS Environments 1
Introduction **1**
Overview of SAS/C CICS Command Language Translator **2**
CICS Background **2**
Basic Considerations for C and CICS **4**

Chapter 2 △ The SAS/C CICS Command Translator 7
Introduction **7**
CICS Command Coding Conventions **7**
How CICS Commands Are Translated **11**
Specifying Translator Options **13**
Translator Option Descriptions **14**

Chapter 3 △ Using C for CICS Application Programs 17
Introduction **17**
SAS/C Considerations **17**
CICS Considerations **20**

Chapter 4 △ Tutorial: Creating a Simple Transaction 29
Introduction **29**
Overview of Processing **29**
The Example Program **30**
Preparing the Example Program **32**
CICS and the Sample Program **33**
Running the Example under CICS **34**

Chapter 5 △ Preprocessing, Compiling, and Linking 35
Introduction **36**
Preprocessing, Compiling, and Linking under TSO **36**
Creating C++ CICS Applications under TSO **40**
Preprocessing, Compiling, and Linking under OS/390 Batch **41**
Creating C++ CICS Applications under OS/390 Batch **47**
Preprocessing, Compiling, and Linking under CMS **50**
Creating C++ CICS Applications under CMS **54**
COOL Options **55**
COOL Messages and CICS Applications **60**
Linking for VSE **61**
Using the External CICS Interface **62**
Using SQL and C **64**
Diagnostics **66**

Chapter 6 △ Running and Debugging SAS/C Programs in the CICS Environment 67

Introduction **67**
Running SAS/C Programs under CICS **67**
Using Run-Time Options **68**
Suggestions for Efficiency **69**
Debugging Your SAS/C CICS Application **69**
Abend Codes **70**

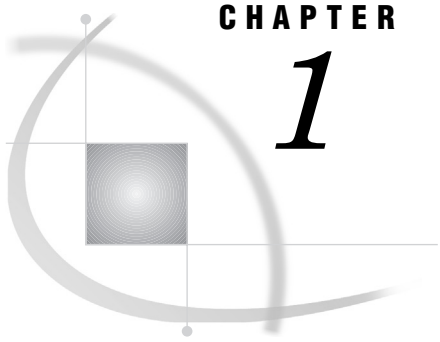
Chapter 7 △ Terminal Control and Basic Mapping Support 71
Introduction **71**
Terminal Control **71**
Basic Mapping Support **72**

Chapter 8 △ Handling Files 77
Introduction **77**
Specifying Filenames and Access Methods **78**
Working with Transient Data **80**
JES Spool File I/O **82**
DL/I Database Support **85**
SQL Database Support **89**

Chapter 9 △ TCP/IP Socket Library Support for the CICS and Environment 91
Overview of TCP/IP **91**
Overview of the BSD UNIX Socket Library **91**
SAS/C Socket Library for TCP/IP **92**
TCP/IP Socket Library Support for CICS **92**
I/O Functions **96**
Unsupported Configuration Information Functions **96**

Appendix 1 △ Examples 99
Introduction **99**
SASCMNU: Display the Main Menu **100**
SASCALL: Perform Inquiry and Update Functions **100**
SASCBRW: Perform Browse Function **109**

Index 119



CHAPTER

1

Overview of the SAS/C and CICS Environments

Introduction **1**
Overview of SAS/C CICS Command Language Translator **2**
CICS Releases Supported **2**
CICS Background **2**
Data-Communication Functions **3**
Data-Handling Functions **4**
Application Program Services **4**
System Services and Monitoring Functions **4**
Basic Considerations for C and CICS **4**
CICS Control Tables and Programs **4**
Re-entrancy **5**
System Architecture **5**
Location of Transient Library **5**
SAS/C Libraries **6**
Systems Programming Environment (SPE) **6**
All-Resident C Programs **6**
Application Design Considerations **6**

Introduction

This book describes the SAS/C CICS Command Language Translator and explains how you can develop CICS command-level application programs in C using the SAS/C Command Language Translator, the SAS/C Compiler, and run-time library.

CICS (Customer Information Control System) is a popular database/data communication (DB/DC) control system offered by International Business Machines Corporation. SAS/C application programs can be developed to take advantage of various CICS services that enable online, real-time transaction processing. Banking and reservation systems are examples of such real-time transaction processing environments. Using the SAS/C Translator, you can develop C programs for a wide range of CICS applications for any system requiring this type of processing.

The SAS/C Compiler is a portable implementation of the high-level C programming language. Using C with CICS is possible because the SAS/C Compiler and Library support the CICS run-time environment–the collection of data and routines that provide interaction with CICS.

This documentation is directed toward experienced C programmers who are familiar with CICS concepts. Experience in applications programming for CICS is helpful, but not required.

Overview of SAS/C CICS Command Language Translator

The SAS/C CICS Command Language Translator enables you to develop application programs under OS/390 or VM and target them to run under CICS. This application-programming interface enables you to request CICS services by placing CICS commands anywhere within your C source code. The SAS/C CICS translator translates these commands into appropriate function calls for communication with CICS.

After the translator translates the CICS commands within your C program, you then compile and link-edit your program as you would any SAS/C program. When you run your SAS/C program, the function calls inserted by the translator invoke the services requested by calling the appropriate CICS control program using the CICS EXEC Interface program.

Chapter 5, "Preprocessing, Compiling, and Linking," on page 35 describes the cataloged procedures, TSO CLISTs, and CMS EXECs that are provided to invoke the translator, compiler, and linker. Chapter 2, "The SAS/C CICS Command Translator," on page 7 describes the compiler and run-time options that are pertinent to the translator.

CICS Releases Supported

Support for both CICS/OS and CICS/DOS from Version 1, Release 7 onward is provided. Currently, this includes the following releases:

- CICS/OS/VS 1.7
- CICS/MVS 2.1
- CICS/ESA 3.1
- CICS/ESA 3.2
- CICS/ESA 3.3
- CICS/ESA 4.1
- CICS Transaction Server for OS/390 1.1
- CICS Transaction Server for OS/390 1.2
- CICS Transaction Server for OS/390 1.3
- CICS/DOS/VSE 1.7
- CICS/DOS/VSE 2.1
- CICS/VSE 2.2
- CICS/VSE 2.3

The SAS/C CICS Command Language Translator has been updated to fully support all new and changed commands through CICS TS for OS/390 Version 1.3, including support for Distributed Program Link and the Front End Programming Interface.

At this time, the SAS/C translator and compiler provides access to CICS only under the OS/390 and VSE family of operating systems. However, you can also prepare SAS/C applications under the VM/SP environment and then run these applications under CICS for the OS/390 or VSE operating systems.

CICS Background

The use of online systems is common today. However, in the past, developing an online system required customized programming involving calls to the operating

system, telecommunication access methods, and data access methods, in addition to application program development.

To free the programmer from concerns about hardware and other components outside the realm of the application itself, Database/Data Communication (DB/DC) control systems were developed. In the late 1960s, CICS was introduced by IBM as one such DB/DC control system. CICS is considered a control system for DB/DC because it provides the control environment for a DB/DC application. That is, an application program that uses databases and data communications in a real-time manner can work with CICS to form a complete DB/DC system.

CICS has evolved from a macro-based DB/DC system to a high-level, command-based language. Early versions of CICS required the programmer to use macros whenever system services were required. Presently, CICS is command-based: one CICS command now accomplishes what used to require a series of macro-based calls. Coding CICS commands in application programs makes requesting CICS services much easier than using earlier macro-based versions. These commands are coded in your SAS/C program using the following general format:

EXEC CICS <*command*>

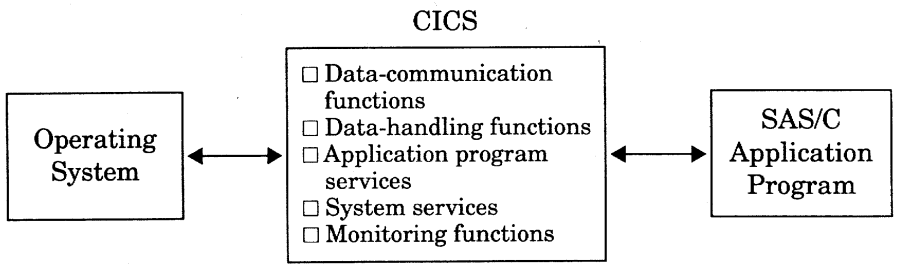
The details of coding commands are discussed in Chapter 2, "The SAS/C CICS Command Translator," on page 7.

As noted earlier, coding CICS commands means that you are requesting CICS services to handle various details of operating system tasks, such as communicating with terminals and storage management. The following components of CICS provide such services to your application program:

- □ data-communications functions
- □ data-handling functions
- □ application program services
- □ system services
- □ monitoring functions.

These functions are described in more detail in the following sections. The relationship between the operating system, CICS, and your application programs is illustrated in Figure 1.1 on page 3.

Figure 1.1 SAS/C CICS Environment



Data-Communication Functions

The CICS services for data communication provide communication interfaces. Communication interfaces can be between other systems and CICS or between terminals and CICS. Data-communication functions include the provision of Basic

Mapping Support (BMS), which frees the application program from concerns with specifics of devices and formats when you are working with displays. Considerations for Basic Mapping Support are discussed in Chapter 7, "Terminal Control and Basic Mapping Support," on page 71.

Data-Handling Functions

This component of CICS provides an interface between CICS and the data needed by your application program, including interfaces with data access methods, such as VSAM, and interfaces with database access methods, such as DL/I. Also, provisions are made for assuring data protection and integrity. Aspects of file handling that are pertinent to your application programs are covered in Chapter 8, "Handling Files," on page 77.

Application Program Services

CICS provides an interface with your program through screen definitions, command interpretation (CECI), debugging facilities, and so forth. These services are described in Chapter 3, "Using C for CICS Application Programs," on page 17.

System Services and Monitoring Functions

The system-service component of CICS provides an interface with the operating system. Functions are provided for program, storage, and task control. Topics related to system services are included in Chapter 3, "Using C for CICS Application Programs," on page 17.

Monitoring functions are provided in the CICS environment for system tuning and performance considerations. Chapter 6, "Running and Debugging SAS/C Programs in the CICS Environment," on page 67 suggests ways of improving performance using C. Refer to the CICS documentation appropriate for your site for additional details.

Basic Considerations for C and CICS

In the chapters that follow, these system components are discussed in terms of steps involved in setting up a SAS/C application program for use under CICS. These steps are discussed in detail in the tutorial in Chapter 4, "Tutorial: Creating a Simple Transaction," on page 29, and again in the comprehensive example in Appendix 1, "Examples," on page 99.

CICS Control Tables and Programs

Developing a basic application program for CICS might consist of writing code to coordinate the flow of control between a terminal, the task at hand, files, and your program. The structure and control necessary for this kind of coordination is provided via CICS system control programs and their associated tables. Files, programs, and transactions must all be defined in the appropriate CICS tables (that is, the FCT, PPT, PCT). For example, providing links to remote systems requires working with a terminal control table (TCT), which is then used by the CICS terminal control program to set up intercommunication.

Whenever appropriate, references may be made to CICS control tables with respect to particular aspects of your SAS/C application program. Examples and discussion of

structuring your program to take advantage of CICS interfaces are found throughout this user's guide.

Re-entrancy

CICS is a multitasking, multithreading system environment. Multitasking means that CICS provides an environment where more than one CICS task runs concurrently. Multithreading means that tasks share the same program in a multitasking environment. CICS is characterized as multitasking and multithreading because it provides an environment in which multiple CICS tasks, using the same program, can run concurrently.

To make this type of environment possible, SAS/C programs running under CICS must be re-entrant. Since, by definition, re-entrant programs do not modify themselves, they can continue processing after an interruption by the operating system. This includes continuing processing after an interruption during which other tasks controlled by the same program may have been executed.

To comply with CICS requirements for re-entrancy, SAS/C programs must be compiled using the **`rent`** (or **`rentext`**) compiler option. Chapter 5, "Preprocessing, Compiling, and Linking," on page 35 provides more information on how to specify options.

System Architecture

You can fully exploit 31-bit addressability. If your program is link-edited as AMODE 31, the library will request that stack, heap, and PRV storage be allocated above the 16-M line. (The PRV, or pseudoregister vector, contains all external and static data for re-entrant programs.) The CICS release and the size of a storage allocation request determine whether the storage is actually allocated above the line. For example, an initial stack size of 3072 bytes is allocated below the line in CICS Version 2, but it is allocated above the line in CICS Version 3. Consult one of the following CICS Application Programmer's Reference manuals for exact details on storage allocation:

CICS/OS/VS, Version 1

CICS/MVS, Version 2

CICS/ESA, Version 3

CICS/ESA, Version 4

CICS Transaction Server for OS/390

SAS/C programs that are link-edited with the specification of AMODE(31) and RMODE(ANY) can be loaded above the 16M line by CICS. The ability for SAS/C programs to reside above the 16M line provides virtual storage constraint relief for storage-constrained CICS systems. That is, programs are not restricted to sharing what below-the-line storage is left over after that used by tables and other components related to CICS.

Location of Transient Library

Most of the SAS/C transient library for CICS can reside above the 16M line. Application programs linked with a release earlier than 6.00 of the resident library will continue to work correctly if they are link-edited as AMODE 31. Programs linked as AMODE 24 must be relinked or have one or more zaps applied. Contact SAS/C Technical Support for information about these zaps. Unless you have a specific reason for doing so, you do not need to link-edit SAS/C CICS applications as AMODE 24.

SAS/C Libraries

The run-time library is freely redistributable, so you can develop your applications and distribute them without interaction with SAS Institute.

Systems Programming Environment (SPE)

The Systems Programming Environment is designed to enable the C language to be used as a systems programming language in the IBM S/390 environment. SPE consists of the minimum number of support routines needed to execute a C program, and a small run-time library that is systems-programming oriented. For complete details on using SPE with CICS, see Chapter 14, "Systems Programming with the SAS/C Compiler," in the SAS/C Compiler and Library User's Guide.

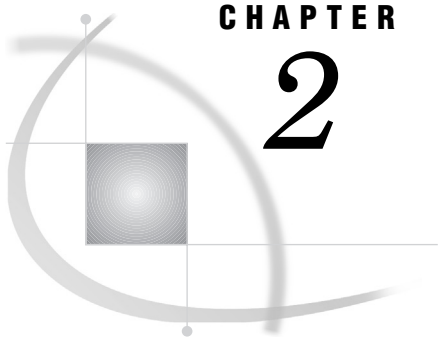
All-Resident C Programs

Normally, when a C program is linked, the resulting load module does not contain all of the support routines needed by the program. These support routines are dynamically loaded from the transient library and released (freeing the memory required as well) when they are no longer needed.

In certain specialized applications and environments, it may be desirable to force the program load module to contain a private copy of all the required support routines. These programs can be characterized as all-resident programs because no transient library routines need to be used. For more information on creating all-resident CICS programs, see Chapter 10, "All-Resident C Programs," in the SAS/C Compiler and Library User's Guide.

Application Design Considerations

Developing application programs for CICS requires attention to design issues involving user-friendliness, system security, and performance.



CHAPTER 2

The SAS/C CICS Command Translator

Introduction 7
CICS Command Coding Conventions 7
Command Format 8
Coding Conventions 8
Commonly Used Data Types 9
Less Common Data Types 9
name 10
label 10
hhmmss 10
cvda 10
Doubleword, Fullword, and Halfword Arguments 10
Character Arguments 11
Prototype Generation 11
How CICS Commands Are Translated 11
Taking Advantage of Prototypes 13
Specifying Translator Options 13
Using the #pragma options Statement 13
Translator Option Descriptions 14

Introduction

The translator accepts and supports all EXEC CICS commands. This includes commands for DL/I, GDS (generalized data stream), DPL (distributed program link), FEPI (front-end programming interface), and BTS (business transaction services). After the initial preprocessing step, the output from the translator is compiled and linked in a way similar to any C program.

This chapter introduces the syntax and conventions for coding CICS commands, explains how the SAS/C CICS Command Translator handles CICS commands coded in your C program, and describes options available for the translator.

CICS Command Coding Conventions

The translator accepts any command or command option that can be used in CICS Transaction Server V1R3. However, the translator does not check to see if a particular command is valid for any given release of CICS. Therefore, if you are programming for an earlier version of CICS, you should be careful to use only those commands that are defined in the version of CICS under which you plan to run your program.

Command Format

The general format of a CICS command is EXEC CICS (or EXECUTE CICS), followed by the name of the function (which may be one, two, or three words), and possibly one or more command options. The entire command must be in uppercase letters. Options may require arguments, or may accept optional arguments. For C applications, CICS commands are terminated by a semicolon.

For example, the original CICS command syntax for the CICS READ command is as follows:

EXEC CICS READ
DATASET(*name*)
FILE(*name*)
INTO(*data-area*)
SET(*prt-ref*)
[LENGTH(*data-area*)]
RIDFLD(*data-area*)
[KEYLENGTH(*data-value*) [GENERIC]]
[SYSID(*name*)]
[RBA | RRN]
[GTEQ | EQUAL]
.
.
.
[UPDATE]

In your C program, this command can be coded as follows:

```
EXEC CICS READ
   DATASET("MASTER")
   INTO(record) LENGTH((short) sizeof(struct INPUT))
   RIDFLD(key) KEYLENGTH(keylen);
```

If you are unfamiliar with the parameters for the CICS READ command, you may want to consult the IBM CICS application programmer's reference appropriate for your site.

Coding Conventions

Observe the following conventions as you write your SAS/C CICS code:

- CICS commands can appear anywhere a left brace can appear.
- The words EXEC CICS (or EXECUTE CICS) must be coded on the same line. There can be no tokens between EXEC and CICS.
- A command can be continued on subsequent lines; however, an option or function name cannot be split across lines (even with a backslash).
- Option arguments can be separated from the option by white space, comments, or new-line characters.
- Option arguments can contain any C expression of the correct type, and may contain C comments. The arguments may be continued on subsequent lines.

□ C comments can be used within CICS commands any place white space can appear.

Note: The translator will not recognize a CICS command in a comment, string literal, or character literal. You should also refrain from coding other constructs the translator would try to translate, such as the following statement:

```
typedef EXEC CICS READ ... ;
```

△

Commonly Used Data Types

Most often, you will use one of these data types, which can follow the options in CICS commands: *data-value*, *data-area* (and *CICS-value data area*), *pointer-value*, and *pointer-ref*. In the context of the C language, these types are used as follows:

data-value
is an expression. *Data-value* arguments provide input only to the command, so they can be constants, expressions, or variables. The expression must have the expected type; that is, if CICS expects a halfword *data-value*, then the expression must be of signed or unsigned short type.

data-area
is an lvalue (specifically, a modifiable lvalue, as specified by the ANSI Standard). *Data-area* arguments may be updated by the CICS command. Therefore, these arguments must be valid objects, for example, variables or storage allocated by the **`malloc`** function.

pointer-value
is any expression of pointer type.

pointer-ref
is a pointer-type lvalue where CICS can store an address. The translator prefixes *pointer-ref* arguments with an ampersand (&). Only *pointer-ref* arguments are modified by the translator.

Pointer-ref
is coded as shown in the following example:

```
EXEC CICS ADDRESS CSA(csa_ptr);
```

The argument **`csa_ptr`** should be declared as a pointer variable, for example:

```
struct CSA *csa_ptr;
```

In this example, the translator will pass **`&csa_ptr`** to CICS. On return from the ADDRESS command, **`csa_ptr`** points to the CSA.

Less Common Data Types

Some less commonly used data types are *name*, *label*, *hhmmss*, and *cvda*:

name
is either a string literal or a pointer (presumably to a string).

label
is always the name of a function (in SAS/C CICS applications).

hhmmss
is an argument for which CICS expects a packed decimal value.

cvda
is a fullword (**int** or **long int**) value always used in the SET and INQUIRE commands.

name

Usually a CICS *name* is required to be of some prescribed length (such as BMS map names, which must be seven characters long). If the name is too short, it must be padded on the right with blanks. If you pass a string literal as a name argument, the translator will compute the length of the string and add padding as necessary, as in the following example:

```
EXEC CICS SEND MAP("ABC") ... ;
```

The translator will pass the following string to the command:

```
"ABC""    "
```

By virtue of C string concatenation, this string is equivalent to "ABC". However, if you pass a pointer to the name, then you are responsible for seeing that the name is blank-padded correctly. Of course, CICS always ignores the trailing '\0' byte at the end. (Because it's past the seventh character, CICS doesn't recognize the trailing '\0'.)

label

The following example uses *label* as the name of a function:

```
EXEC CICS HANDLE CONDITION ERROR(err_func);
```

In this example, **err_func** defines an error handling routine in your C program. See Chapter 3, "Using C for CICS Application Programs," on page 17 for more details.

hhmmss

For *hhmmss*, the translator expects a pointer to an aggregate object, such as a character array or structure, because C has no equivalent to packed decimal data. (See the descriptions for **pdset** and **pdval** in SAS/C Library Reference, Volume 1.)

cvda

The *cvda* data type expects a fullword argument; therefore, you can use an **int** or a **long**, **signed** or **unsigned**. In the SET command, you can use a constant integer or the result of the DFHVALUE function, although using one of the command options is a more likely choice. In the INQUIRE command, use an lvalue.

Doubleword, Fullword, and Halfword Arguments

CICS sometimes describes arguments as *doubleword*, *fullword* or *halfword*. In C terms, a doubleword is a **signed** or **unsigned long long**. A fullword is a **signed** or **unsigned int**, or a **signed** or **unsigned long**. A halfword is a **signed short**. You can use **unsigned shorts**, but the compiler will copy it to a temporary and pass the address of the temporary to CICS. This could cause a problem if the variable in question is passed to CICS and then assigned a value (for example, record length). In such a case, CICS will update the temporary, which is probably not what the program is expecting.

Also, if a CICS argument accepts a string literal (names, output lines, and so on), the translator accepts the same string literals that the compiler does. This means that you can use concatenated string literals (for example, "ABC" "DEF") and strings with escape sequences (for example, "\xc1 \xc2 \xc3").

Character Arguments

When a CICS command option expects a single character as an argument, you must use a **char** pointer argument rather than a character variable. If the argument is a data value, you can also use a string literal. For example, the DATESEP option of the FORMATTIME command expects a character to be used to separate the parts of a formatted date. The methods used in either of the following examples can be used to code this correctly:

Example Code 2.1 Example 1

```
char *separator = ":";
EXEC CICS FORMATTIME ... DATESEP(separator);
```

Example Code 2.2 Example 2

```
EXEC CICS FORMATTIME ... DATESEP(":");
```

When you use the PROTO translator option (the default) with a **char** variable or **char** literal, a prototype mismatch message results.

Prototype Generation

By default, the translator generates a prototype for each CICS command. If the translator expects a C pointer argument, it uses **void*** in that argument's place in the function prototype. Since **void*** is the generic pointer type, you can use any pointer type you want for the argument. For instance, a character pointer and a structure pointer are equally acceptable when writing to transient data queues with the WRITEQ TD command using the FROM option, as in the following example:

```
char *output_buffer;
struct FORMATTED_LINE *line;

EXEC CICS WRITEQ TD FROM(output_buffer) ... ;
EXEC CICS WRITEQ TD FROM(line) ... ;
```

How CICS Commands Are Translated

CICS C programs generally do not call a C library function to request operating system services. Instead, the programmer codes a CICS command to issue the request. The translator translates these commands into calls to the CICS EXEC interface program. This program provides the requested service by invoking the appropriate CICS control program.

The translator performs the following actions when it begins translating the CICS commands in your program:

- it includes the **<cics.h>** header file at the top of the output file.
- the original CICS commands remain in the file, but they are surrounded with **#if 0 / #endif**.
- the command is translated into
  - a left brace

- a function pointer that is declared for prototype generation
- a right brace.

For example, the following program fragment illustrates how a call to read a record in a VSAM file would be coded using the READ CICS command:

```
/* before translation */
void readin(struct INPUT *record,
  char *key, short keylen)
{
.
.
.
   EXEC CICS READ DATASET("MASTER")
        INTO(record) LENGTH(sizeof(struct INPUT))
        RIDFLD(key) KEYLENGTH(keylen);
.
.
.
}
```

During translation, the READ command is converted into the following C statements:

```
/* after translation */
#include <cics.h>

void readin(struct INPUT *record,
   char *key, short keylen)
{
.
.
.
   #if 0
   EXEC CICS READ DATASET("MASTER")
        INTO(record) LENGTH(sizeof(struct INPUT))
        RIDFLD(key) KEYLENGTH(keylen);
   #endif
   {
   __ref void (*_ccp_exec_cics)(char *,
    const char *,void *,short,void *,
   short)= (__ref void (*)(char *,const char *,
    void *,short,void *,short))
   _ccpexec;_ccp_exec_cics
     ("\x06\x02\xf8\x00\x09\x00\x00\x80\x00"
       "00000006",
   "MASTER" "  ",record,sizeof(struct INPUT),
      key,keylen);
   }
.
.
.
}
```

First, the **cics.h** header file is included at the top of the output file. Next, the original command remains in the source file, but is surrounded with the **#if 0**/ **#endif**. CICS commands are then translated into the following sequence (assuming the PROTO option is in effect):

1. a left brace.
2. a function pointer declaration, **_ccp_exec_cics**. (This pointer is assigned a pointer to the EXEC command interface function **_ccpexec** that is declared in **cics.h**. This declaration generates a prototype. Because the translator is not able to determine whether the types of the function arguments are correct, the prototypes allow the compiler to check them.)
3. a call to the EXEC interface function.
4. a right brace.

The first argument to the EXEC command interface (\x06\x02 . . .) is a bit string containing encoded information about the type of command, the number of arguments, and whether or not the arguments are dummy arguments. The "00000006" sequence is the number of the line on which the command started. The remaining arguments are the C variables and constants specified in the command. The entry point to the EXEC command interface function used and the bits set in the first argument change according to the command and the options in effect.

Taking Advantage of Prototypes

The translator automatically generates prototypes for CICS command arguments. Prototype generation is useful because it helps determine whether the argument types are correct. For example, this feature enables you to catch mistakes such as specifying an **int** type variable instead of a **short**.

The translator also pads short strings in commands. CICS doesn't recognize null-terminated strings, which helps to prevent errors such as incorrectly identifying a program. For example, the EXEC CICS LOAD command could be coded as follows:

```
EXEC CICS LOAD PROGRAM("FTOC");
```

After translation, the command appears as the following:

```
EXEC CICS LOAD PROGRAM("FTOC" "    ");
```

Specifying Translator Options

You can specify options to modify the way the translator handles your program, just as you can specify compile-time and linkage options. Specify translator options as you would regular compiler options, except there are no short-form options. For example, this means you cannot code **-p** for PAGESIZE.

Using the #pragma options Statement

A subset of translator options may be used in a special options statement embedded in the source file. The options, described in the next section, are specified by using the **#pragma options** statement. For example, you can code the following preprocessing directive:

```
#pragma options xopts(dli,cics)
```

The **#pragma options** statement may appear anywhere a **#pragma** statement can appear. The set of current option settings may be saved and restored by using the **#pragma options push xopt** and **#pragma options pop xopts** statements, as in the following example:

```
/* Save the current setting of    */
/* the PROTO option.              */
#pragma options push xopts

/* Suppress prototype generation  */
/* for this command.              */
#pragma options xopts(noproto)

EXEC CICS ... ;

/* Restore the previous setting   */
/* of the PROTO option.           */
#pragma options pop xopts
```

Chapter 5, "Preprocessing, Compiling, and Linking," on page 35 provides additional information on specifying options for the translator.

Translator Option Descriptions

The following is a list of the options that are accepted by the translator. If the option may be specified with **`#pragma options xopts`**, it is marked with an asterisk (*).

CICS*
is the default. It processes EXEC CICS commands.

CBMSMAPS
tells LCCCP0 that the BMS maps were generated specifically for C language programs, which is supported in CICS/ESA 3.3 and later. The translator generates different default values for the FROM option of the SEND MAP command and the TO option of the RECEIVE MAP command depending on the presence (or absence) of this option. If the TO and FROM options are always explicitly coded, this option has no effect. The default is NOCBMSMAPS.

COMNEST*
accepts nested comments.

DEBUG*
is the default. It produces code for the Execution Diagnostic Facility (EDF).

DLI*
processes EXEC DLI commands.

EDF*
allows interception of all commands by the EDF.

Note: Even when you use the EDF option, you will not see the library's issuance of any of the commands issued by the library on your behalf because the library uses NOEDF. △

EXPAND*
shows C code generated for commands in the source listing.

FILES(*xxx*)
(OS/390 only) replaces the SYS prefix in translator DD names with *xxx*. The value of *xxx* may be no more than three characters in length. The first character must be alphabetic or either @, #, or $. The second and third characters must be alphanumeric or either @, #, or $.

FLAG(*x*)*
emits messages only of level *x* and above. The value of *x* may be either I (notes), W (warnings), E (errors), or S (severe errors). FLAG (I) is the default.

JAPAN*
results in uppercased C keywords (**`VOID`**, **`INT`**, and so on) in the command translations. This is intended for use with the compiler's **`japan`** option.

OPTIONS
is the default. It lists the translator options in effect.

OUTLRECL(*nnn*)
(CMS only) specifies the LRECL of the translator output file. The value of *nnn* may range from 40 to 255, inclusive. The default is 255.

OUTRECFM(*x*)
(CMS only) specifies the RECFM of the translator output file. The value of *x* may be either F or V. The default is V.

OUTSEQ(*n*,*m*)
adds sequence numbers to the output file. The value of *n* specifies the first sequence number and *m* the incrementing value. If both *n* and *m* are 0, then the output file is not sequenced. The default is OUTSEQ(0,0).

OVERSTRIKE
prints special characters in the listing file as overstrikes.

PAGESIZE(*n*)*
defines the number of lines per page in the listing file. The default is PAGESIZE(60).

PRINT
produces a listing file. Under TSO, the PRINT option specifies the data set name of the listing file. Under OS/390, the listing file is written to SYSPRINT. Under CMS, the listing fileid is specified by the PR(fileid) option of the LCCCP EXEC.

PROTO*
is the default. It generates a prototype for **`_ccp_exec_cics`** for each call to the EXEC command interface function.

SOURCE*
is the default. It includes a formatted source listing in the listing file.

TERM
directs diagnostic messages to the terminal. TERM is the default under CMS and under TSO. Under OS/390 NOTERM is the default in batch mode.

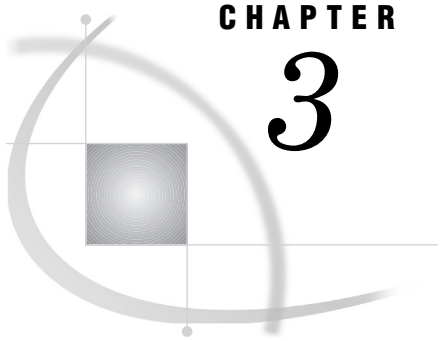
TRANS
translates special characters to their listing file representations.

TRIGRAPHS*
accepts ANSI Standard trigraphs in the input file and uses them in the translated output.

UPPER
prints lowercase letters as uppercase in the listing file, and uses uppercase letters in diagnostic messages. UPPER implies OVERSTRIKE.

XREF*
includes in the listing file a cross-reference of all the commands used in the input file.

For details on all standard options provided with the SAS/C compiler, consult the SAS/C Compiler and Library User's Guide.



CHAPTER

3

Using C for CICS Application Programs

Introduction **17**
SAS/C Considerations **17**
Header Files for SAS/C CICS Applications **18**
Indep Compiler Option **18**
SAS/C Library Functions **18**
CICS Considerations **20**
Data Declarations for CICS **21**
Using Global External Variables **21**
BMS Definitions **23**
Supplying Arguments to main( ) **23**
Handle Condition and Handle AID **24**
HANDLE CONDITION **24**
HANDLE CONDITION and recursion **25**
HANDLE AID **25**
Abend and Error Handling **25**
SAS/C I/O Functions and CICS **26**
Environment Variable Support **26**
Program Control **26**
Interval and Task Control **26**
Debugging SAS/C CICS Applications **26**

Introduction

This chapter provides information you need to begin writing SAS/C programs for execution in the CICS environment. There are two main sections: SAS/C considerations (for example, header files and library support) and CICS considerations (for example, how CICS sees your application program).

SAS/C Considerations

This section discusses header files and library functions supported for SAS/C applications under CICS.

Header Files for SAS/C CICS Applications

Several header files are associated with a SAS/C CICS application. These header files are contained in SASC.MACLIBC under OS/390, and LC370 MACLIB under CMS. The following header files are included automatically by the translator:

`cics.h`
contains constants and declarations needed by the translator.

`dib.h`
maps the DL/I interface block.

`eiblk.h`
maps the execution interface block (EIB).

The following header files are also available for inclusion by the programmer:

`dfhaid.h`
provides the standard attention identifier definitions.

`dfhbmsca.h`
lists the standard attribute and printer control character definitions.

`dfhcdblk.h`
maps the conversation data areas for EXEC CICS GDS commands.

`dfhmsrca.h`
lists the magnetic slot reader control value constant definitions.

`dfh2980.h`
maps the CICS 2980 General Banking Terminal System constants.

`dliuib.h`
maps the DL/I user interface block (UIB).

Indep Compiler Option

SAS/C programs can be compiled freely with the **`indep`** compiler option. Use of this option causes the generation of code that can be called before the C framework is initialized, or code that can be used for interlanguage communication.

When the **`indep`** option is used, the L$UPREP provided by SAS/C stores the CRAB address in the first word of the Transaction Work Area (TWA) so that CICS application programs will be re-entrant; this is a CICS requirement. A transaction that invokes a C program compiled with the **`indep`** option must be defined with a TWA. See Appendix 6, "Using the indep Option for Interlanguage Communication," and Chapter 14, "Systems Programming with the SAS/C Compiler" in the SAS/C Compiler and Library User's Guide for more details.

SAS/C Library Functions

The majority of the functions in the SAS/C Library are supported. The following categories of functions are supported (any exceptions are noted):

- input/output
- dynamic loading
- signal handling (synchronous only)
- coprocessing

- memory allocation
- diagnostic control
- compatibility
- inline machine code
- program control
- timing functions (except **clock**)
- general utility
- localization/multibyte character functions
- math functions
- string utility functions
- character type macros
- system interface functions (except **oslink** and **system**).
- varying-length argument list functions
- TCP/IP socket library (except for configuration information functions)

The library functions in the following categories are not supported:

- file management functions
- UNIX style I/O functions (except **isatty**)
- IUCV functions
- CMS low-level I/O
- OS/390 low-level I/O
- SUBCOM interfaces to CLISTs and EXECs
- REXX interface functions
- ILC functions
- OS/390 multitasking
- APPC/VM
- other SAS/C POSIX functions

Table 3.1 on page 19 contains a complete listing of all functions that are not supported under CICS.

Table 3.1 Functions Not Supported by CICS

Function Category	Unsupported Function Names
Timing	**clock**
System Interface	**getlogin** **oslink** **system**

Function Category	Unsupported Function Names
I/O	**aopen** **close** **closedir** **creat** **ctermind** **dup** **dup2** **_fcntl** **fdopen** **fsync** **ftruncate** **getdtablesize** **kdelete** **kgetpos** **kinsert** **kreplace** **kretrv** **ksearch** **kseek** **ktell** **lseek** **_lseek** **open** **_open** **opendir** **pclose** **pipe** **popen** **read** **_read** **readdir** **rewinddir** **tmpfile** **tmpname** **ttyname** **write** **_write**
Signal Handling	**alarm** **alarmd** **ecbpause** **kill** **oesigsetup** **pause** **sigpause** **sigsuspend**

Note: See Table 9.1 on page 97 for a list of the unsupported functions in the TCP/IP socket category. △

CICS Considerations

This section provides a quick, overall look at how C interfaces with commonly used CICS components and features. Additional information is provided in subsequent chapters. In general, CICS sees SAS/C programs as assembler language programs. Therefore, SAS/C application programs are defined as assembler language programs to CICS in the processing program table (PPT).

Data Declarations for CICS

The following declarations may be needed, depending on your application:

- execution interface block (EIB) definitions
- basic mapping support (BMS) attribute definitions
- DL/I support (DIB, DL/I header files).

When you use the SAS/C CICS Translator, the EIB and DIB are automatically included in your program. Include the header files for BMS and DL/I only if you are using those functions.

The header files necessary for these declarations were noted earlier in "Header Files for SAS/C CICS Applications" on page 18. The following sections introduce you to working with these components using C.

Using Global External Variables

CICS application programs typically require access to various CICS control blocks, notably the EIB, so that the programs can check return code values and so on. The SAS/C implementation provides several global external variables to address the most commonly referenced areas. This means you can bypass using either the ADDRESS EIB or the ADDRESS command to obtain control block information.

Note: In this context, *global* means these external variables are also shared with dynamically loaded modules. △

The following global external variables are provided:

__commptr
points to the CICS COMMAREA (otherwise NULL).

__dibptr
points to the DL/I interface block.

__eibptr
points to the EIB.

The following examples illustrate how to use these variables.

Example Code 3.1 Using __eibptr

```
.
.
.
switch(_eibptr->EIBRESP){
   case DFHRESP(NORMAL):
      break;
   case DFHRESP(NOTOPEN):
      return -1;
   default:
.
.
.
}
```

Example Code 3.2 Defining your own __eibptr

```
.
.
.
struct EIBLK *my_eib_ptr;
EXEC CICS ADDRESS EIB(my__eib_ptr);
.
.
.
switch(my_eib_ptr->EIBRESP) {
.
.
.
}
```

Example Code 3.3 Using __commptr

```
if (__commptr) {
/* Because __commptr was not null, */
/* a COMMAREA was passed           */
/* Perform "second" pass           */
/* processing.                     */
   EXEC CICS RECEIVE MAP...
.
.
.
}
else {
/* Because __commptr was null,     */
/* no COMMAREA was passed;         */
/* therefore, this must be         */
/* the "first" pass.               */
/* Display the main menu.           */
      EXEC CICS SEND MAP...
.
.
.
}
```

Example Code 3.4 Using __dibptr

```
.
.
.
#define STATUS(code)

(memcmp(__dibptr->DIBSTAT,code,2)==0)
.
.
```

.

BMS Definitions

The BMS map language must be specified as assembler. The symbolic map (DSECT) must be processed by DSECT2C using this special compiler option: **-d**. To assist with this process under OS/390, the SAS/C Compiler and Library provides a DSECT2C cataloged procedure. Chapter 7, "Terminal Control and Basic Mapping Support," on page 71 provides complete documentation for SAS/C BMS considerations.

Supplying Arguments to main( )

The C **main** function can be passed arguments optionally. For CICS, the default arguments to **main** are similar to other traditional high-level languages under CICS, that is, a pointer to the execution interface block and a pointer to any COMMAREA (or a NULL pointer, if no COMMAREA exists). These arguments are supplied by the run-time library to **main** as an alternative to using the CICS global external variables described earlier in this chapter. To use this technique, code your program as follows:

```
main(struct EIBLK *eib_pointer,
void *COMMAREA_pointer)
{
.
.
.
}
```

This is the default behavior by the run-time library if the linkage editor control statement ENTRY MAIN is used when linking your program. If a SAS/C program is invoked via a CALL statement to the MAIN entry point from another language, the CALL statement must pass pointers to the EIB and COMMAREA as the two parameters.

Alternatively, if you need to pass other parameters besides those of the EIB and COMMAREA, you can use the alternate entry points $MAINC and $MAINO. These entry points are primarily for use if your **main** program is invoked via a CALL statement from another language. When you use these entry points, the arguments to **main** are the more traditional **argc** and **argv** values. The **argv** vector is defined as follows:

```
/* pointer to transaction id  */
argv [0] char *tranid
/* pointer to the EIB         */
argv [1] struct EIB *ptr
/* pointer to any COMMAREA    */
argv [2] void *COMMAREA
/* pointer to optional parm 1 */
argv [3] void *parm1
.
.
.
/* pointer to optional parm n */
argv [n] void *parmn
```

When coding the CALL statement from another language, you can code the CALL statement specifying $MAINC or $MAINO directly. Again, the CALL statement must pass the EIB and COMMAREA pointers as the first two parameters followed,

optionally, by other parameters. Note that the C run-time library processes this parameter list as a VL-style parameter list and does not convert the CICS pseudo-null COMMAREA pointer value of x'ff000000' to a NULL pointer. A pointer value of x'ff000000' always indicates the end of the parameter list. If you are passing optional parameters, you must pass a value of 0 if no COMMAREA is being passed to the called program. Because the library expects a VL-style parameter list when called through the entry points $MAINC/$MAINO, failing to pass one could cause the **argc** value to be incorrect or lead to other program failures.

For example, here is a short piece of COBOL code that calls the statically linked C routine named **called1**:

```
PROCEDURE DIVISION.
 CALL 'CALLED1' USING DFHEIBLK DFHCOMMAREA.
 EXEC CICS RETURN END-EXEC.
 GOBACK.
```

And the following is representative of the way you can code the C routine **called1**:

```
void called1()
{
   EXEC CICS ASKTIME;
}
```

The entry points $MAINC and $MAINO can also be used when a **main** program is invoked directly by CICS (either as the first program of a transaction or via a LINK or XCTL command). Select the entry point via the linkage editor control statement ENTRY $MAINC or ENTRY $MAINO. Note that ESA versions of CICS do not always construct VL-style parameter lists when calling programs directly; thus, you may get unpredictable results.

For more information about the entry points $MAINC and $MAINO, see Chapter 11, "Communication with Assembler Programs," in the SAS/C Compiler and Library User's Guide.

When the initial C function invoked by CICS is compiled with the **indep** compiler option, the arguments can have any type and, therefore, cannot be modified, or even inspected, by the library. In this case, **_commptr** is always set to NULL, **argc** is 0, and **argv** is NULL. For more information, see the SAS/C Compiler and Library User's Guide.

Handle Condition and Handle AID

Under CICS, your SAS/C program is responsible for handling conditions raised by CICS for each CICS command executed from your application. This means your program should test for a normal response; otherwise, a default system action occurs. You can test for a normal response by coding the RESP option on any CICS command, thereby testing the response code directly. Alternatively, you can code the HANDLE CONDITION and HANDLE AID commands as illustrated in the following sections.

HANDLE CONDITION

The standard syntax for this command is

```
EXEC CICS HANDLE CONDITION condition(label) ...
```

;

The SAS/C implementation is

```
EXEC CICS HANDLE CONDITION condition(function-name) ... ;
```

Here is an example:

```
EXEC CICS HANDLE CONDITION ERROR(errfunc);
   .
   .
   .
   void errfunc(int errcode)
   {
   .
   .  /* errcode is EIBRESP */
   .
   }
```

Normal return from the function is to the instruction immediately after the EXEC command that raised the condition. Note that your program can freely execute a **`longjmp()`** from the error handler.

HANDLE CONDITION and recursion

When you code the error-handling function, be careful not to issue a CICS command that could possibly raise an error condition. This could lead to recursion, for example, to an infinite loop or short-on-storage condition.

To avoid the possibility of recursion within the error-handling function, you can revert the error handling back to CICS with the following CICS command:

```
EXEC CICS HANDLE CONDITION ERROR;
```

Because no function is specified, you avoid the potential for recursion.

HANDLE AID

The syntax for this command is

```
EXEC CICS HANDLE AID option(label) ... ;
```

The SAS/C implementation is

```
EXEC CICS HANDLE AID option(function-name) ... ;
```

Here is an example:

```
EXEC CICS HANDLE AID PF1(help);
.
.
.
void help(int aid)
{
.
/* AID values are mapped in <dfhaid.h> */
.
.
}
```

Standard return from the function is to the instruction immediately after the EXEC command that caused the AID to be transmitted to CICS.

Abend and Error Handling

All the error-handling, tracebacks, and messages you expect from the SAS/C Library are fully supported, including

- signal handling
- tracebacks
- warning and error messages.

However, the label option of the EXEC CICS HANDLE ABEND command, coded as follows, is not supported:

```
EXEC CICS HANDLE ABEND LABEL(label);
```

If the label option is coded, the message LSCP038 is generated.

If your program issues any other variant of the command, it will completely override the run-time library's abend handling. In other words, any of the three following commands would override the run-time library's abend handling:

```
EXEC CICS HANDLE ABEND PROGRAM(name);
EXEC CICS ABEND CANCEL;
EXEC CICS PUSH HANDLE;
```

All abnormal termination messages are written to the transient data destination SASE. The output is prefixed with standard terminal and transaction identifier information.

SAS/C I/O Functions and CICS

Except for UNIX style I/O functions, all other SAS/C Library I/O functions for sequential input and output are supported for CICS applications. C also provides features for working with transient data and JES Spool File input and output, as well as support for DL/I and SQL. Chapter 8, “Handling Files,” on page 77 provides details.

Environment Variable Support

With Release 6.00 of SAS/C, certain functions have been enhanced to support environment variables under CICS. Consult Chapter 4, "System Interface Functions and Environment Variables," and the function descriptions for **getenv** and **putenv** in SAS/C Library Reference, Volume 1 for details on this support.

Program Control

The CICS commands LOAD, RELEASE, LINK, RETURN, and XCTL are fully supported. COMMAREA is also fully supported and may be allocated anywhere that is convenient: the stack or the heap. Note that use of the XCTL or RETURN commands will terminate the C environment before execution of the command. This means that control will not return to the C program.

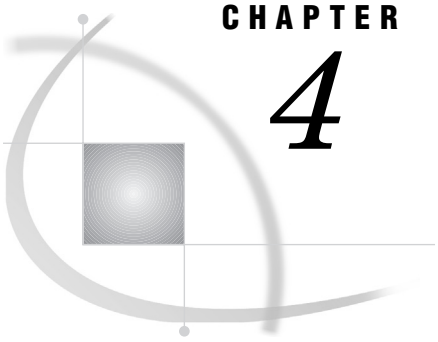
Interval and Task Control

The only SAS/C consideration in this area is to note that the **sleep** function is fully supported and recommended instead of DELAY. This is because **sleep** accepts an **int** argument rather than a packed decimal.

Debugging SAS/C CICS Applications

You can use the SAS/C Debugger with CICS in a remote session to debug your SAS/C application programs. This is documented in the SAS/C Debugger User's Guide and

Reference. Also, the Execution Diagnostic Facility (EDF) is fully supported. Facilities such as CA-INTERTEST are also available, but no symbolic debugging is currently supported; that is, all debugging is done in assembler language mode.



CHAPTER

4

Tutorial: Creating a Simple Transaction

Introduction **29**
Overview of Processing **29**
The Example Program **30**
Preparing the Example Program **32**
Using TSO under OS/390 **32**
Using OS/390 Batch **32**
CICS and the Sample Program **33**
Running the Example under CICS **34**

Introduction

Read this chapter if you are familiar with the CICS environment but not with C as the application program language, or if you are familiar with both CICS and C and would like a quick introduction to using C under CICS.

After reading this chapter, you should be able to preprocess the example program using the SAS/C CICS Command Language Translator, compile and link the example, and then initiate the example transaction under CICS.

Note: This program, and more extensive examples of how to code SAS/C programs for use under CICS (see Appendix 1, "Examples," on page 99) can be found in the programs contained in the SASC.SAMPLE library under OS/390 and LCSAMPLE MACLIB under CMS. See your site representative for details on how to access these programs at your installation. △

Although an overview of the CICS processing involved in the example is provided, this chapter assumes you understand the basics of CICS applications.

Overview of Processing

CICS enables online, real-time processing of requests for work, called *transactions*. These transactions can work with a database, or they can be requests for information, such as system information supplied by CICS. The sample program in this chapter focuses on how to use the SAS/C Translator. The CICS portion of the example defines a transaction that requests information from CICS. The system used to process the example is OS/390; however, the steps are similar whether you use TSO, OS/390 batch, CMS, or the SAS/C Cross-Compiler product (see Chapter 5, "Preprocessing, Compiling, and Linking," on page 35).

The Example Program

The example program (SASCSAMP) is designed to produce a formatted report of the SAS/C transient programs used within CICS. The CICS INQUIRE command is used to identify all programs beginning with LSH (the default identifier for SAS/C transient programs). The CICS SEND TEXT command is used to output the report to the CICS terminal screen. Screen details and paging are processed automatically by the SEND command. Because the SEND TEXT command is used, no mapping steps are needed. An error-handling routine is also coded to illustrate how C employs the CICS HANDLE CONDITION command.

The example program follows:

```
/*- - - - - - - - - - - - - - - - - - - - - - +
|                                             |
|           S A S / C  S A M P L E            |
|                                             |
|          NAME: SASCSAMP                     |
|       PURPOSE: Sample program used in CICS  |
|                Tutorial chapter             |
|  INSTALLATION: Follow the instructions in   |
|                the Tutorial chapter of the  |
|                SAS/C CICS User's Guide.     |
|       COMPILE:                              |
|       EXECUTE:                              |
|         USAGE:                              |
| SYSTAEM NOTES:                              |
|                                             |
+ - - - - - - - - - - - - - - - - - - - - -  */

  /*- - - - - - - - - - - - - - - - - - - */
  /* This sample program produces a list  */
  /* of all the CICS programs that begin  */
  /* with the string ''lsh''.             */

#pragma options xopts(xref)

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

main()
{
   int  next_resp = 0;
   char header [ ] = "This is a list of SAS/C
    transient programs\n";
   short hdrlen = sizeof(header) - 1;
   char pgmname [8];
   extern char msg [80];
   extern short msglen;
   void errhndlr();

   EXEC CICS HANDLE CONDITION ERROR(errhndlr);

   EXEC CICS SEND TEXT FROM(header)
```

```
      LENGTH(hdrlen) ACCUM PAGING;

   EXEC CICS INQUIRE PROGRAM START;

   while(next_resp == 0){
    EXEC CICS INQUIRE PROGRAM(pgmname)
     NEXT RESP(next_resp);
      if (memcmp(pgmname, "LSH", 3))
      continue;
      msglen = sprintf(msg, "   %.8s\n", pgmname);
      EXEC CICS SEND TEXT FROM(msg)
       LENGTH(msglen) ACCUM PAGING;
   }

   EXEC CICS INQUIRE PROGRAM END;
   EXEC CICS SEND PAGE;
}

   /* Something has gone wrong.*/
   /* Send a message to give a clue why. */
void errhndlr(int errcode)
{
   char savefn [2]; /* area for saving EIBFN */
   memcpy(savefn, _eibptr->EIBFN, 2);

      /* Make sure there's no recursion. */
   EXEC CICS HANDLE CONDITION ERROR;
    /* Purge any partial messages. */
    EXEC CICS PURGE MESSAGE;

   msglen = sprintf(msg,
    "CICS Function %02x%02x received
     error code %d",
        savefn[0], savefn[1], errcode);

   EXEC CICS SEND TEXT FROM(msg)
    LENGTH(msglen) ERASE;
   exit(-1);
}
```

If you are familiar with other application languages under CICS, you'll notice after looking over this code that C function names are given as arguments to EXEC command options instead of as program labels. For example, in the following code, **`errhndlr`** refers to a SAS/C error-handling function:

```
EXEC CICS HANDLE CONDITION ERROR(errhndlr);
```

For more details on this and other SAS/C coding conventions see "CICS Command Coding Conventions" on page 7. The program also provides an example of including options by using **`#pragma options`**.

The **`main`** function defines variables for the output heading, program name, and message, as well as a prototype for the error-handling routine.

CICS commands are then used to do the following:

- handle attention identifiers (HANDLE CONDITION ...)
- send data without mapping (SEND TEXT ...)
- begin query of CICS system's data (INQUIRE PROGRAM START ...).

As long as there are transient programs to be listed, the program continues, requesting CICS to display the results on the terminal. The program terminates by ending the query process (INQUIRE PROGRAM END ...) and sending the last page of data (SEND PAGE ...).

The error-handling function **errhndlr()** is used to catch processing errors and print a message instead of relying on default CICS processing. See Chapter 3, "Using C for CICS Application Programs," on page 17 for more details on handling error conditions.

Preparing the Example Program

To run the example under CICS, you must first use the SAS/C CICS translator and the SAS/C Compiler and Library to preprocess, compile, and link-edit the source code. You can perform these steps under OS/390, VM, or with the SAS/C Cross-Compiler product under UNIX. In this section, we use TSO under OS/390 and OS/390 batch to illustrate how to prepare the example so you can run it under CICS.

Using TSO under OS/390

Using TSO, you execute a CLIST that invokes the translator, then proceed with compilation and linking as you would with any SAS/C program.

Because the translator is itself a C program, you must ensure that the transient run-time library is allocated to the DDname CTRANS or is installed in the system link list. Your installation will probably cause it to be allocated automatically. Consult your SAS Software Representative for SAS/C software products to determine if this has been done. If not, use the TSO ALLOCATE command to associate the library with the CTRANS DDname, as shown in the following example:

```
ALLOC FI(CTRANS) DA('SASC.LINKLIB') SHR
```

To translate the SAS/C example program, you execute the LCCCP CLIST. Specify the input data set as the SAS/C sample library, and specify the output data set as one of your own user data sets, as shown in the following example:

```
LCCCP 'SASC.SAMPLE(SASCSAMP)' OUTPUT(your.data-set)
```

Next, you execute the regular SAS/C compilation CLIST, LC370. Specify as input the same data set as the one used for OUTPUT in the previous step. If you are not familiar with these steps using the SAS/C compiler, see the SAS/C Compiler and Library User's Guide for additional information on executing SAS/C CLISTs. In particular, see the sections on TSO execution styles.

After you have preprocessed and compiled the example program, you are ready to execute the regular SAS/C link-edit CLIST, CLK370. Specify as input the data set that contains the output from the LC370 CLIST. Specify, via the LOAD option, the name of your CICS program library. Specify the CICS option so that CICS resident-routines are included at link time.

Refer to Chapter 5, "Preprocessing, Compiling, and Linking," on page 35 and to the SAS/C Compiler and Library User's Guide for more information on how to execute SAS/C CLISTs.

Using OS/390 Batch

Under OS/390 batch, you first build a job stream that executes the cataloged procedure LCCCPCL. Allocate the SASC.SAMPLE library to the SYSIN DD of the

translator step (CCP). The PDS member name of the example program is SASCSAMP. Allocate your CICS program library to the SYSLMOD DD of the link-edit step (LKED). An example follows:

```
//jobname  JOB ...
// EXEC LCCCPCL
//CCP.SYSIN  DD DSN=SASC.SAMPLE(SASCSAMP),DISP=SHR
//LKED.SYSLMOD DD DSN=your.cics.loadlib(SASCSAMP),
   DISP=SHR
```

You're now ready to submit the job stream for execution. All return codes will be 0 if the translation, compilation, and link-edit have been successful. Refer to Chapter 5, "Preprocessing, Compiling, and Linking," on page 35 and to the SAS/C Compiler and Library User's Guide for more information on how to execute SAS/C cataloged procedures.

CICS and the Sample Program

CICS sees the example program as follows:

1. Terminal control reads the data entered and passes control to task control.
2. A task is created and validated against the program control table (PCT).
3. If all is valid, program control then loads the associated example application program to process the task.
4. Control is then passed to the example application program.
5. The example application program terminates when processing is complete, and CICS passes control back to task control.

To prepare the example program to run under CICS you must first ensure that the library in which the executable load modules were stored is available on the CICS DFHRPL DD. The local CICS system programmer or the SAS Software Representative for SAS/C software products can be of help in locating this library.

Next, you need to define a transaction ID in the PCT for the example program. This may be accomplished by using RDO (resource definition online, transaction CEDA) or DFHCSDUP. For older releases of CICS, table-generation macros may be used. The DFHCSDUP input is used here because it is easiest to understand. Contact your local CICS system programmer for assistance. Example DFHCSDUP input follows:

```
DEFINE TRANSACTION(SASC) GROUP(your-group-name)
       DESCRIPTION(SAS/C Example transaction)
       PROGRAM(SASCSAMP)
```

Finally, you need to define a program name in the PPT for the example program in a manner similar to the way you defined the transaction. Example DFHCSDUP input follows:

```
DEFINE PROGRAM(SASCSAMP) GROUP(your-group-name)
       DESCRIPTION(SAS/C Example Program)
       LANGUAGE(ASSEMBLER)
```

SAS/C application programs are defined as assembler language programs to CICS.

After defining the transaction and program (using RDO or DFHCSDUP), you must make them known to CICS by installing them (using the INSTALL command) with the CEDA transaction. If you have used the older table-generation method, then CICS will need to be restarted so that the definitions will be recognized.

Running the Example under CICS

After you complete the SAS/C and CICS steps discussed in the previous sections, you will be ready to run the example under CICS. To try the sample transaction, type the transaction identifier **SASC** in the upper left corner of the display and press ENTER.

CHAPTER 5

Preprocessing, Compiling, and Linking

Introduction **36**
Preprocessing, Compiling, and Linking under TSO **36**
Files Used by the Translator **36**
The LCCCP CLIST **37**
Compiling SAS/C Programs for CICS under TSO **38**
Linking SAS/C Programs for CICS under TSO **39**
When to Use COOL **39**
Linking All-Resident Programs **39**
Using TSO CLISTs to Link **39**
Executing CLK370 without the IBM Linkage Editor **40**
Creating C++ CICS Applications under TSO **40**
The LCCCP CLIST **40**
The LCXX CLIST **40**
The COOL CLIST **41**
Preprocessing, Compiling, and Linking under OS/390 Batch **41**
Using LCCCP to Translate C Programs **41**
Using LCCCPC to Compile C Programs **42**
Using Cataloged Procedures to Compile and Link C Programs **43**
Selecting the Entry Point **43**
Selecting the Program Environment **43**
Creating All-Resident Load Modules **44**
Using LCCCPCL to Preprocess, Compile, and Link C Programs **44**
Using LCCCL to Link SAS/C Programs **45**
Creating C++ CICS Applications under OS/390 Batch **47**
LCCPCXX **47**
LCCCXXL **49**
LCCPCXXL **50**
LCCPCXXA **50**
Preprocessing, Compiling, and Linking under CMS **50**
The Translator Input File **51**
The Translator Output File **51**
The Listing File **51**
Terminal Output **51**
The LCCCP EXEC **51**
Compiling SAS/C Programs for CICS under CMS **52**
Linking SAS/C Programs for CICS under CMS **52**
When to Use COOL **52**
Linking All-Resident Programs **53**
Using CMS EXECs to Link **53**
Creating C++ CICS Applications under CMS **54**
The LCCCP EXEC **54**

The LCXX EXEC **54**
COOL Options **55**
COOL Messages and CICS Applications **60**
Linking for VSE **61**
Linking CICS/VSE Applications under CMS **61**
Linking CICS/VSE Applications under OS/390 **62**
Creating the VSE Phase **62**
Using the External CICS Interface **62**
The EXEC CICS Interface **63**
Running your CICS Application with the External CICS Interface **63**
Translating, Compiling, and Linking under OS/390 **63**
Using TSO **63**
Using OS/390 Batch **64**
Translating, Compiling, and Linking under CMS **64**
Using SQL and C **64**
Diagnostics **66**

Introduction

This chapter describes how to preprocess, compile, and link your SAS/C CICS application program under the following environments:

- TSO
- OS/390 batch
- VM/CMS.

COOL options and messages relevant to CICS applications are also described. Additional sections include guidelines for linking applications that will be ported to a VSE environment, and a sample cataloged procedure for using the SAS/C CICS translator with the SQL preprocessor. The types of diagnostics provided by the translator are also described. Chapter 6, “Running and Debugging SAS/C Programs in the CICS Environment,” on page 67 explains how to run, test, and debug your program.

Preprocessing, Compiling, and Linking under TSO

The following sections describe how to use the translator (LCCCP) interactively under TSO. The translator alone may be invoked by using the LCCCP CLIST. Alternatively, you can choose to run the translator with the compiler, with the compiler and the linkage editor, or with the compiler with COOL and the linkage editor.

Files Used by the Translator

Depending on which options you specify, you will need two or more of the following files to use the translator under OS/390:

- SYSIN
- SYSPUNCH
- SYSPRINT
- SYSTERM.

Following are descriptions of each of these files and when they are required. Under TSO, you specify the data set by using CLIST options.

SYSIN
is always required; it describes a C source file containing embedded CICS commands. The input data set can have either fixed-length or variable-length records, blocked or unblocked. LCCCP places no restriction on the LRECL of this data set.

The input data set may have sequence numbers. The translator checks the first record in the source file to determine whether the source file has sequence numbers. If the source file has varying length records, the translator inspects columns 1 through 8; otherwise, it inspects the last eight columns. If the translator finds a sequence number in the first record, the corresponding columns of all subsequent records are ignored.

SYSPUNCH
is always required; it describes the output data set that contains the translated C source file. The output data set can have either fixed-length or variable-length records, blocked or unblocked.

LCCCP places no restriction on the LRECL of this data set, but the LRECL should be at least that of the input data set. If the input data set has fixed-length records and the output data set has variable-length records, then the output LRECL should be at least four characters longer. If the OUTSEQ option is used and the input data set does not have sequence numbers, the output LRECL should be at least eight characters longer.

Note: The compiler will not accept an input file containing records longer than 1024 characters. △

SYSPRINT
is required if the PRINT option is used; it describes the data set to which the listing will be printed. The listing data set should have a logical record length of 121 and a record format of FBA, and should be a sequential data set.

The listing contains a list of the LCCCP options in effect (if the OPTIONS option is used), a source listing (if the SOURCE option is in effect), and a cross-reference listing (if the XREF option is in effect). Diagnostic messages are also written to the listing. If the EXPAND option is used, the translation of each CICS command is shown.

SYSTERM
is optional; it describes a data set that is used for LCCCP diagnostic messages (if the TERM option is used) and for C library warning messages. The DCB requirements are supplied by LCCCP. This file is automatically allocated by default under TSO.

The LCCCP CLIST

When you use TSO to prepare a SAS/C application program to run in the CICS environment, you invoke a CLIST for the translator before continuing with SAS/C CLISTs for compiling and linking. For information on the SAS/C CLISTs for compiling and linking, see the SAS/C Compiler and Library User's Guide.

To begin the preprocessing step under TSO, invoke the LCCCP CLIST. Because the translator is itself a C program, you must ensure that the transient run-time library is allocated to the DDname CTRANS or is installed in the system link list. Your installation will probably cause it to be allocated automatically. Consult your SAS Software Representative for SAS/C software products to determine if this has been

done. If not, use the TSO ALLOCATE command to associate the library with the CTRANS DDname as shown in the following:

```
ALLOC FI(CTRANS) DA('SASC.LINKLIB') SHR
```

The format of the LCCCP CLIST is
LCCCP *in-dsname* [OUTPUT(*out-dsname*)]
[PRINT(*print-file*)] *options*
where

in-dsname
is the name of the SYSIN data set. If the data set belongs to another user, the fully qualified name of the data set must be enclosed in three apostrophes. If the data set name is not fully qualified, the LCCCP CLIST adds the user's prefix and a final qualifier of CCP, if necessary.

out-dsname
is the name of the SYSPUNCH data set. If the data set belongs to another user, the fully qualified name of the data set must be specified and enclosed in three apostrophes. If the data set name is not fully qualified, the LCCCP CLIST adds the user's prefix and a final qualifier of C.

If *out-dsname* is specified and is not fully qualified, the CLIST will use *out-dsname* with a final qualifier of C.

print-file
is the name of the SYSPRINT data set. This data set must be sequential. If not fully qualified, the CLIST adds the user's prefix and a final qualifier of CCPLIST.

options
are any LCCCP options, as shown in "Specifying Translator Options" on page 13.

The following is an example of this command:

```
lcccp 'userid.cics.source(example)'
      output(cout(example))
```

In this example, the C source code is contained in the PDS referenced by ***'userid.cics.source(example)'***. The output data set SYSPUNCH is referred to by ***output(cout(example))***. Because no options are specified, the default values will be used in the translation.

Compiling SAS/C Programs for CICS under TSO

When you execute the SAS/C compilation CLIST, LC370, specify as input the same data set as the one you used for output in the preprocessing step. If you are not familiar with the SAS/C Compiler, see the SAS/C Compiler and Library User's Guide for additional information on executing SAS/C CLISTs. In particular, consult the sections on TSO execution styles.

Note: CICS requires that all programs be compiled so that they are re-entrant. You must specify the compiler options **rent** or **rentext** to cause the compiler to generate re-entrant code because the compiler default for re-entrancy is **norent**. △

For example, to compile the file FTOC.C.A, you issue the following command:

```
LC370 ftoc (rent)
```

Linking SAS/C Programs for CICS under TSO

After you have preprocessed and compiled your program, you are ready to execute the SAS/C link-edit CLIST, CLK370. Specify as input the data set that contains the output from the LC370 CLIST. Use the LOAD option to specify the name of your CICS program library, and use the CICS option so that CICS resident routines will be included at link time.

Regardless of the linking method you use, you must always

- include the CICS Execution Interface stub routines in each load module
- arrange for the stub to be the very first thing in the load module.

The CLISTs distributed by SAS Institute have been written to automate this process by using the following linkage-editor control statements in one form or another:

```
LIBRARY DFHLIB(DFHEAI,DFHEAIO)
ORDER   DFHEAI
```

The DDname DFHLIB points to the CICS load library that contains the execution interface stub routines.

When to Use COOL

You must use COOL to preprocess your object code if one or more of the following conditions apply:

- two or more compilations in the program are compiled using either of the compiler options (**rent** or **rentext**)
- the program initializes external variables in two or more re-entrant compilations
- you use the all-resident library
- you specify the **EXTNAME** option for more than one compilation.

See “Using TSO CLISTs to Link” on page 39 and the SAS/C Compiler and Library User’s Guide for more information on COOL.

Linking All-Resident Programs

The default name of the all-resident library is SASC.CICS.ARESOBJ. (Ask your SAS Software Representative for SAS/C software products for the name of the library at your site.) When linking an all-resident program, concatenate the all-resident library in front of any other autocall data sets, and include the object deck created by compiling a source file that includes **<resident.h>** and the appropriate macro definitions. See SAS/C Compiler and Library User’s Guide for more information on linking all-resident programs. The process of linking all-resident programs is automated through the use of keywords and parameters in the cataloged procedures, CLISTs, and EXECs.

Using TSO CLISTs to Link

The CLK370 CLIST invokes the COOL object code translator, followed by the linkage editor. Optionally, you can skip the COOL step by specifying the NOCOOL option. The format is as follows:

CLK370 *dsname* <*keywords*>

where *dsname* is the name of the object data set that is to be the primary input to COOL or the linkage editor. The data set name should be the name of the data set containing the object code, or the COOL/linkage-editor control statements used as input, or both. Follow standard TSO naming conventions; that is, if the data set belongs to

another user, the full name of the data set must be specified, and the name must be enclosed in three apostrophes. If the object code is in a member of a partitioned data set, the member name must be specified in parentheses following the data set name in the normal TSO manner. The final qualifier of the input data set name is assumed to be OBJ. If you do not add this qualifier, it is supplied automatically by the CLIST.

Keywords indicate COOL options, linkage-editor options, or the names of other data sets to use during linking. You must specify either the **CICS** or the **CICSVSE** keyword when linking CICS applications. See "COOL Options" on page 55 for more information.

Note: The resident library for the SAS/C compiler is divided into a base resident library and the resident library for standard environments. The base resident library, SASC.BASEOBJ, contains system-independent code. The standard CICS resident library, SASC.CICSOBJ, contains system-dependent code. For more details about using the CLK370 CLIST, consult the SAS/C Compiler and Library User's Guide. △

Executing CLK370 without the IBM Linkage Editor

CLK370 accepts a NOCOOL option that causes the linkage editor to be invoked directly without using the COOL utility. CLK370 allows you to specify any linkage-editor options, such as **LIST**, **LET**, **MAP**, **XREF**, **TEST**, **RENT**, **OVLY**, **AMODE**, and **RMODE**. (These options are valid for the linkage editor whether or not COOL is run.) The IBM linkage editor and loader manual discusses these options.

Creating C++ CICS Applications under TSO

You must execute these three separate CLISTs to create a C++ CICS application under TSO:

LCCCP	invokes the CICS translator.
LCXX	invokes the C++ translator and compiler.
COOL	invokes the SAS/C prelinker and linkage editor to generate a load module.

The LCCCP CLIST

In the LCCCP CLIST, use an input data set with a final qualifier of CPP. The output data set name is specified with the **output** option; the final qualifier should be C. In the following example, the output data set name is fully qualified because the data set name has a different final qualifier (CXX) than the one expected by the LCXX CLIST. Use standard TSO naming conventions to specify all data set names.

Assuming an input data set named USERID.CICS.CPP and an output data set named USERID.CICS.CXX, the LCCCP CLIST can be invoked as follows:

```
LCCCP CICS(SAMPLE)
 OUTPUT('''USERID.CICS.CXX(SAMPLE)''')
```

The LCXX CLIST

In the LCXX CLIST, the final qualifier of the input and output data sets should be CXX and OBJ, respectively. Assuming an input data set named USERID.CICS.CXX

and an output data set named USERID.CICS.OBJ, the LCXX CLIST can be invoked as follows using the **object** option:

```
LCXX CICS(SAMPLE) OBJECT(CICS(SAMPLE)) RENT
```

The **object** option specifies the data set in which the output from the C++ translator will reside. Use standard TSO naming conventions to specify the data set names if qualifiers other than those noted are used.

Note: Always compile CICS applications with the **RENT** compiler option.

For additional information on the LCXX CLIST, refer to SAS/C Cross-Platform Compiler and C++ Development System User's Guide. △

The COOL CLIST

The COOL CLIST invokes the SAS/C prelinker and the linkage editor; the final qualifier of the input data set should be OBJ. Use the **LOAD** option to specify the data set in which the linkage editor stores the output load module. If the **LOAD** option is not specified, various rules apply as to how this data set will be determined. Consult Chapter 7, "Linking C Programs," in the SAS/C Compiler and Library User's Guide for additional information on these rules.

Assuming an input data set of USERID.CICS.OBJ and an output data set name of USERID.CICS.LOAD, the COOL CLIST can be invoked as follows:

```
COOL CICS(SAMPLE) CICS CXX RENT LOAD(CICS(SAMPLE))
```

The **CXX** and **CICS** options are needed to add the required libraries to COOL's autocall list. Use standard TSO naming conventions to specify the data set names if qualifiers other than those noted here are used.

Note: CICS applications must always be link-edited with the **RENT** option. △

Preprocessing, Compiling, and Linking under OS/390 Batch

Under OS/390 batch, you use the following cataloged procedures to compile and link your program under CICS:

LCCCP
: to preprocess

LCCCPC
: to preprocess and compile

LCCCPCL
: to preprocess, compile, and link.

Using LCCCP to Translate C Programs

The LCCCP cataloged procedure may be used to execute the translator. The JCL contained in this procedure is similar to that shown in the following:

```
//        EXEC LCCCP,PARM.CCP='options'
//SYSPUNCH DD DISP=SHR,DSN=your.translated.source(member)
//SYSIN    DD DISP=SHR,DSN=your.source.library(member)
```

When you use LCCCP, you need to provide DD cards for only SYSIN (your C source program containing EXEC CICS commands) and SYSPUNCH (where the translated C

source file is placed). The LCCCP procedure contains the JCL shown in Example Code 5.1 on page 42.

Example Code 5.1 Expanded JCL for LCCCP

```
//LCCCP   PROC
//*
//CCP      EXEC PGM=LCCCP0,REGION=1536K
//STEPLIB  DD DSN=SASC.LINKLIB,DISP=SHR
//         DD DSN=SASC.LOAD,DISP=SHR
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSPUNCH DD UNIT=SYSDA,DSN=&&CCPOUT,DISP=(,PASS),SPACE=(TRK,(5,5)),
//            DCB=(RECFM=VB,LRECL=259)
```

Using LCCCPC to Compile C Programs

You can use the LCCCPC procedure to preprocess and then compile your C program. (Under TSO and CMS, preprocessing, compiling, and linking are typically done separately.) Complete details about compiler options and running the compiler in different environments are given in the SAS/C Compiler and Library User's Guide.

Note: CICS requires that all programs be compiled so that they are re-entrant. You must specify the compiler options **rent** or **rentext** to cause the compiler to generate re-entrant code because the compiler default for re-entrancy is **norent**. Static variables cannot be modified if you compile with the **rentext** option. △

The procedure LCCCPC for OS/390 batch runs the translator in one step, followed by an invocation of the compiler. The JCL for preprocessing and then compiling C programs using LCCCPC is as follows:

```
//        EXEC LCCCPC, PARM.C='RENT'
//CCP.SYSIN  DD DISP=SHR,DSN=your.source.library(member)
//C.SYSLIN DD DISP=SHR,DSN=your.object.library(member)
```

When you use LCCCPC, you need to provide DD statements for only SYSIN (your C source program containing EXEC CICS commands) and SYSLIN (your object data set). The LCCCPC procedure contains the JCL shown in Example Code 5.2 on page 42.

Example Code 5.2 Expanded JCL for LCCCPC

```
//LCCCPC PROC
//*
//CCP      EXEC PGM=LCCCP0,REGION=1536K
//STEPLIB  DD DSN=SAS.LINKLIB,
//            DISP=SHR RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
//            DISP=SHR TRANSLATOR LIBRARY
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSPUNCH DD UNIT=SYSDA,DSN=&&CPPOUT,DISP=(NEW,PASS),
//            SPACE=(TRK,(5,5)),DCB=(RECFM=VB,LRECL=259)
//C        EXEC PGM=LC370B,PARM='RENT',COND=(8,LT,CCP)
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
//            DISP=SHR COMPILER LIBRARY
```

```
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT2   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT3   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSLIN   DD DSN=&&OBJECT,SPACE=(3200,(10,10)),DISP=(MOD,PASS),
//            UNIT=SYSDA
//SYSLIB   DD DSN=SASC.MACLIBC,
//            DISP=SHR STANDARD MACRO LIBRARY
//SYSDBLIB DD DSN=&&DBGLIB,SPACE=(4080,(20,20,1)),DISP=(,PASS),
//            UNIT=SYSDA,DCB=(RECFM=U,BLKSIZE=4080)
//SYSTMP01 DD UNIT=SYSDA,SPACE=(TRK,25)      VS1 ONLY
//SYSTMP02 DD UNIT=SYSDA,SPACE=(TRK,25)      VS1 ONLY
//SYSIN    DD DSN=*.CCP.SYSPUNCH,DISP=(OLD,DELETE,DELETE),
//            VOL=REF=*.CCP.SYSPUNCH
```

Using Cataloged Procedures to Compile and Link C Programs

Two cataloged procedures are provided for preprocessing, compiling, and linking, or simply linking a C program for CICS: LCCCPCL and LCCCL. The LCCCPCL procedure invokes the linkage editor; the LCCCL procedure invokes COOL before invoking the linkage editor.

Selecting the Entry Point

You must explicitly specify the entry point of the program in a linkage-editor ENTRY control statement. When using either LCCCPCL or LCCCL, you can specify (via the ENTRY parameter) an ENTRY control statement to be added to the input automatically. Valid values of the ENTRY parameter are as follows:

ENTRY=MAIN
: links a program whose entry is a C main program. MAIN is the default value.

ENTRY=CSPE
: links CICS SPE applications. It specifies the standard CICS start-up routine as the entry point.

ENTRY=DYN
: links subordinate load modules (which are dynamically loaded by some other module at run-time via the **`loadm`** function).

ENTRY=NONE
: is used in any situation where a special entry point is needed. This value inhibits the inclusion of an ENTRY control statement. You must add an ENTRY control statement to the input that specifies the correct entry point.

Selecting the Program Environment

The ENV symbolic parameter may be used to specify the environment in which the program is to run. Valid values of the ENV parameter are as follows:

ENV=CICS
: specifies the default CICS environment.

ENV='CICS.SPE'
: specifies that this application will run in the CICS Systems Programming Environment.

ENV=VSE
specifies that this application should be linked with code to enable it to run in a CICS/VSE environment. This parameter is used only with the LCCCCL procedure. For further information, see “Linking for VSE” on page 61.

Creating All-Resident Load Modules

The ALLRES symbolic parameter may be used to force the program load module to contain a private copy of all the required transient library routines. This is a special-use feature not normally specified by applications developers. This parameter is used only with the LCCCCL procedure. For more information, see Chapter 10, "All-Resident C Programs," in the SAS/C Compiler and Library User's Guide. Valid values of the ALLRES parameter are as follows:

ALLRES=NO
specifies that normally transient library routines will not be included in the load module. This is the default.

ALLRES=YES
forces the transient library routines to be included in the load module.

Using LCCCPCL to Preprocess, Compile, and Link C Programs

The procedure LCCCPCL invokes the translator and the compiler, followed by the linkage editor. The following is the JCL for preprocessing, compiling, and then linking C programs using LCCCPCL:

```
//        EXEC LCCCPCL,PARM.CCP='options', PARM.C='RENT'
//CCP.SYSIN DD DISP=SHR,DSNAME=your.source.library(member)
//LKED.SYSLMOD DD DISP=SHR,DSNAME=your.cics.loadlib(member)
```

When you use LCCCPCL, you need to provide DD statements for only SYSIN (your C source program) and SYSLMOD (your CICS application load module library). See “Using Cataloged Procedures to Compile and Link C Programs” on page 43 for a complete description of the ENV= and ENTRY= JCL parameters. The LCCCPCL procedure contains the JCL shown in Example Code 5.3 on page 44.

Example Code 5.3 Expanded JCL for LCCCPCL

```
//LCCCPCL PROC ENTRY=MAIN,ENV=CICS,
//         CALLLIB='SASC.BASELIB',
//          SYSLIB='SASC.BASELIB',
//         CICSLIB='CICS17.LOADLIB'
//*
//*************************************************************
//* ENV=CICS:       MODULE RUNS IN A CICS C ENVIRONMENT       *
//* ENV='CICS.SPE': MODULE RUNS IN A CICS C SPE ENVIRONMENT   *
//* ENTRY=MAIN:     MODULE IS A NORMAL C MAIN PROGRAM         *
//* ENTRY=DYN:   MODULE IS DYNAMICALLY LOADABLE AND REENTRANT *
//* ENTRY=NONE:     ENTRY POINT TO BE ASSIGNED BY USER        *
//* ENTRY=CSPE:     MODULE IS A CICS SPE APPLICATION          *
//*************************************************************
//*
//CCP     EXEC PGM=LCCCP0,REGION=1536K
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR C RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
```

```
//            DISP=SHR C COMPILER LIBRARY
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*,
//            DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSPUNCH DD UNIT=SYSDA,DSN=&&CPPOUT,DISP=(NEW,PASS),
//            SPACE=(TRK,(5,5)),DCB=(RECFM=VB,LRECL=259)
//C       EXEC PGM=LC370B,PARM='RENT',COND=(8,LT,CCP)
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR C RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
//            DISP=SHR C COMPILER LIBRARY
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT2   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT3   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSLIN   DD DSN=&&OBJECT,SPACE=(3200,(10,10)),DISP=(MOD,PASS),
//            UNIT=SYSDA
//SYSLIB   DD DSN=SASC.MACLIBC,
//            DISP=SHR STANDARD MACRO LIBRARY
//SYSDBLIB DD DSN=&&DBGLIB,SPACE=(4080,(20,20,1)),DISP=(,PASS),
//            UNIT=SYSDA,DCB=(RECFM=U,BLKSIZE=4080)
//SYSTMP01 DD UNIT=SYSDA,SPACE=(TRK,25)      VS1 ONLY
//SYSTMP02 DD UNIT=SYSDA,SPACE=(TRK,25)      VS1 ONLY
//SYSIN    DD DSN=*.CCP.SYSPUNCH,DISP=(OLD,DELETE,DELETE),
//            VOL=REF=*.CCP.SYSPUNCH
//*
//LKED    EXEC PGM=LINKEDIT,PARM='LIST,MAP,RENT',
//             COND=((8,LT,C),(8,LT,CCP))
//SYSPRINT DD SYSOUT=*,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=*
//SYSLIN   DD DSN=*.C.SYSLIN,DISP=(OLD,PASS),VOL=REF=*.C.SYSLIN
//         DD DSN=SASC.CICSOBJ(EP@&ENTRY),
//            DISP=SHR
//         DD DDNAME=SYSIN
//SYSLIB   DD DSN=SASC.&ENV.LIB,
//            DISP=SHR               CICSLIB
//         DD DSN=&SYSLIB,DISP=SHR   COMMON RESIDENT LIBRARY
//         DD DSN=&CALLLIB,DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,50))
//SYSLMOD  DD DSN=&&LOADMOD(MAIN),DISP=(,PASS),UNIT=SYSDA,
//            SPACE=(1024,(50,20,1))
//DFHLIB   DD DSN=&CICSLIB,DISP=SHR CICS APPLICATION STUBS
```

Using LCCCL to Link SAS/C Programs

LCCCL invokes COOL and the linkage editor. Typical JCL for running this cataloged procedure is shown in the example that follows. This example shows two separate process/compilation steps, followed by the step to link the application together.

```
//STEP1  EXEC LCCCPC
//CCP.SYSIN  DD DISP=SHR,DSN=your.source.library(mainpgm)
//C.SYSLIN   DD DSIP=SHR,DSN=your.object.library(mainpgm)
```

```
   .
   .
   .
//STEP2  EXEC LCCCPC
//CCP.SYSIN  DD DISP=SHR,DSN=your.source.library(subpgm)
//C.SYSLIN   DD DSIP=SHR,DSN=your.object.library(subpgm)
   .
   .
   .
//LINKSTEP  EXEC LCCCL
//SYSLMOD   DD DSIP=SHR,DSN=your.cics.loadlib(applname)
//SYSIN     DD DISP=SHR,DSN=your.object.library(mainpgm)
//          DD DISP=SHR,DSN=your.object.library(subpgm)
```

The JCL steps (STEP1 and STEP2) shown must specify only two DD statements: SYSIN (the C source program) and SYSLIN (the resulting object code library). These steps can be run independently from any of the others.

The LINKSTEP JCL step specifies the DD statements for SYSLMOD (the CICS application load module library) and SYSIN (the collective object code to be processed). This step lets the ENV= and ENTRY= parameters take their default values (of CICS and MAIN, respectively).

The LCCCL procedure specifies the DD statement for SYSLKCTL. This DD statement is used to pass input to the linkage editor from the COOL batch monitor, CLK370B. The statement also contains linkage-editor control statements needed by applications running under CICS. The LCCCL procedure contains the JCL shown in Example Code 5.4 on page 46.

Example Code 5.4 Expanded JCL for LCCCL

```
//LCCCL      PROC ENV=CICS,ALLRES=NO,ENTRY=MAIN,
//             CALLLIB='SASC.BASEOBJ',
//             SYSLIB='SASC.BASEOBJ'
//*
//* ************************************************************
//* ENV=CICS:       MODULE RUNS IN A CICS C ENVIRONMENT        *
//* ENV=VSE:        MODULE RUNS IN A CICS/VSE ENVIRONMENT      *
//* ENV='CICS.SPE': MODULE RUNS IN A CICS C SPE ENVIRONMENT    *
//* ENTRY=MAIN:  MODULE IS A NORMAL C MAIN PROGRAM             *
//* ENTRY=DYN:   MODULE IS DYNAMICALLY LOADABLE                *
//* ENTRY=CSPE:  MODULE IS A CICS SPE APPLICATION              *
//* ENTRY=NONE:  ENTRY POINT TO BE ASSIGNED BY USER            *
//* ************************************************************

//LKED     EXEC PGM=CLK370B,PARM='LIST,MAP,RENT',REGION=1536K
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR C COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR C RUNTIME LIBRARY
//SYSPRINT DD SYSOUT=*,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=*
//SYSLIN   DD UNIT=SYSDA,DSN=&&LKEDIN,SPACE=(3200,(20,20)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLKCTL DD DSN=SASC.CICSOBJ(EP@&ENTRY),
//            DISP=SHR
//         DD DSN=*.SYSLIN,VOL=REF=*.SYSLIN,
//            DISP=(SHR,PASS)
```

```
//SYSLIB   DD DDNAME=AR#&ALLRES       ARESOBJ OR ENVIRONMENT OBJ FILE
//         DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR                 ENVIRONMENT SPECIFIC OBJECT FILE
//         DD DSN=&SYSLIB,DISP=SHR     COMMON RESIDENT LIBRARY
//         DD DSN=&CALLLIB,DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,50))
//SYSLMOD  DD DSN=&&LOADMOD(MAIN),DISP=(,PASS),UNIT=SYSDA,
//            SPACE=(1024,(50,20,1))
//AR#NO    DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR
//AR#YES   DD DSN=SASC.CICS.ARESOBJ,
//            DISP=SHR
//DFHLIB   DD DSN=CICS17.LOADLIB,
//            DISP=SHR
```

Creating C++ CICS Applications under OS/390 Batch

This release of the SAS/C CICS command language translator provides four cataloged procedures to facilitate the creation of C++ CICS applications under OS/390:

LCCPCXX
: invokes the CICS translator, C++ translator, and compiler.

LCCCXXL
: invokes COOL (SAS/C's prelinker) and the linkage editor.

LCCPCXXL
: invokes the CICS translator, the C++ translator, the compiler, COOL, and the linkage editor in a single step.

LCCPCXXA
: invokes the CICS translator, the C++ translator, the compiler, and the AR370 archive utility in a single step.

Here's an example of the JCL necessary to invoke the first two cataloged procedures listed above:

```
//COMPILE    EXEC LCCPCXX,PARM.CCP='CICS options',PARM.X='C++ options'
//CCP.SYSIN    DD DSN=your.source.library(member),DISP=SHR
//X.SYSLIN     DD DSN=your.object.library(member),DISP=SHR
//*
//LINK       EXEC LCCCXXL,PARM.LKED='COOL and LINK EDIT options'
//LKED.SYSLMOD DD DSN=your.cics.load(member),DISP=SHR
//LKED.SYSIN   DD DSN=your.object.library(member),DISP=SHR
```

LCCPCXX

You must provide two DD statements when you invoke the LCCPCXX cataloged procedure:

SYSIN	your C++ source program that contains **`EXEC CICS`** commands.
SYSLIN	your object library.

Note: You must always compile CICS applications with the **RENT** option; you can specify this option in the PARM.X option list. △

The LCCPCXX procedure contains the JCL shown in Example Code 5.5 on page 48.

Example Code 5.5 Expanded JCL for LCCPCXX

```
//LCCPCXX    PROC
//*************************************************************
//*  NAME:  LCCPCXX                                 (LCCPCXX)    ***
//*  PRODUCT:  SAS/C++                                           ***
//*  PROCEDURE:  INVOKE CICS/VS COMMAND LANGUAGE TRANSLATOR,     ***
//*              C++ TRANSLATOR, COMPILER                        ***
//*  DOCUMENTATION:  SAS/C++ TRANSLATOR USER'S GUIDE             ***
//*  FROM:  SAS INSTITUTE INC.                                   ***
//*         SAS CAMPUS DRIVE                                     ***
//*         CARY, NC 27513                                       ***
//*************************************************************
//*
//CCP      EXEC PGM=LCCCP0,REGION=1536K
//STEPLIB  DD DSN=SASC.LINKLIB,
//            DISP=SHR C RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
//            DISP=SHR C COMPILER LIBRARY
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*,
//            DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSPUNCH DD UNIT=SYSDA,DSN=&&CCPOUT,DISP=(,PASS),
//            DCB=(RECFM=VB,LRECL=259),
//            SPACE=(TRK,(5,5))
//*
//X        EXEC PGM=LC370CX,PARM='RENT'
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR TRANSLATOR LIBRARY
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSTROUT DD DSN=&&TROUT,SPACE=(6160,(10,10)),DISP=(NEW,PASS),
//            UNIT=SYSDA
//SYSIN    DD DSN=*.SYSTROUT,VOL=REF=*.SYSTROUT,DISP=(OLD,PASS)
//SYSUT1   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT2   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT3   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSLIN   DD DSN=&&OBJECT,SPACE=(3200,(10,10)),DISP=(MOD,PASS),
//            UNIT=SYSDA,DCB=(RECFM=FB,LRECL=80)
//SYSLIB   DD DSN=SASC.MACLIBC,
//            DISP=SHR C/C++ STANDARD HEADER FILES
//SYSDBLIB DD DSN=&&DBGLIB,SPACE=(4080,(20,20,1)),DISP=(,PASS),
//            UNIT=SYSDA,DCB=(RECFM=U,BLKSIZE=4080)
//SYSTMP01 DD UNIT=SYSDA,SPACE=(TRK,25)      VS1 ONLY
//SYSTMP02 DD UNIT=SYSDA,SPACE=(TRK,25)      VS1 ONLY
//SYSTRIN  DD DSNAME=*.CCP.SYSPUNCH,DISP=(OLD,DELETE,DELETE),
//            VOL=REF=*.CCP.SYSPUNCH
```

LCCCXXL

You must provide two DD statements when you invoke the LCCCXXL cataloged procedure:

SYSIN
your object library.

SYSLMOD
your CICS application load module library.

Note: CICS applications must always be link-edited with the **RENT** option; you can specify this option in the PARM.LKED option list. △

The LCCCXXL procedure contains the JCL shown in Example Code 5.6 on page 49.

Example Code 5.6 Expanded JCL for LCCCXXL

```
//LCCCXXL   PROC ENV=CICS,ALLRES=NO,ENTRY=MAIN,
//            CALLLIB='SASC.BASEOBJ',
//            SYSLIB='SASC.BASEOBJ',
//            CICSLIB='CICS330.SDFHLOAD',
//            CXXLIB='SASC.A'
//*************************************************************
//*  NAME:  LCCCXXL                                (LCCCXXL)    ***
//*  PRODUCT:  SAS/C++                                          ***
//*  PROCEDURE:  CICS C++ APPLICATION CLINK                     ***
//*  DOCUMENTATION:  SAS/C++ TRANSLATOR USER'S GUIDE            ***
//*  FROM:  SAS INSTITUTE INC.                                  ***
//*         SAS CAMPUS DRIVE                                    ***
//*         CARY, NC 27513                                      ***
//*************************************************************
//*************************************************************
//* ENV=CICS:        MODULE RUNS IN A CICS ENVIRONMENT
//* ENV=VSE:         MODULE RUNS IN A CICS/VSE ENVIRONMENT
//* ENV='CICS.SPE':  MODULE RUNS IN A CICS SPE ENVIRONMENT
//* ENTRY=MAIN:      MODULE IS A NORMAL C MAIN PROGRAM
//* ENTRY=DYN:       MODULE IS DYNAMICALLY LOADABLE
//* ENTRY=CSPE:      ENTRY POINT IS CICS SPE STANDARD START-UP ROUTINE
//* ENTRY=NONE:      ENTRY POINT TO BE ASSIGNED BY USER
//*************************************************************
//*
//LKED     EXEC PGM=COOLB,PARM='RENT,LIST,MAP',REGION=1536K
//STEPLIB  DD DSN=SASC.LOAD,
//            DISP=SHR C COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
//            DISP=SHR C RUNTIME LIBRARY
//SYSPRINT DD SYSOUT=*,DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=*
//SYSLIN   DD UNIT=SYSDA,DSN=&&LKEDIN,SPACE=(3200,(20,20)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLKCTL DD DSN=SASC.CICSOBJ(EP@&ENTRY),
//            DISP=SHR
//         DD DSN=*.SYSLIN,VOL=REF=*.SYSLIN,
//            DISP=(SHR,PASS)
```

```
//SYSLIB   DD DDNAME=AR#&ALLRES       ARESOBJ OR ENVIRONMENT OBJ FILE
//         DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR                 ENVIRONMENT SPECIFIC OBJECT FILE
//         DD DSN=&SYSLIB,DISP=SHR     COMMON RESIDENT LIBRARY
//         DD DSN=&CALLLIB,DISP=SHR
//SYSARLIB DD DSN=&CXXLIB,DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,50))
//SYSLMOD  DD DSN=&&LOADMOD(MAIN),DISP=(,PASS),UNIT=SYSDA,
//            SPACE=(1024,(50,20,1))
//AR#NO    DD DSN=SASC.&ENV.OBJ,
//            DISP=SHR
//AR#YES   DD DSN=SASC.CICS.ARESOBJ,
//            DISP=SHR
//DFHLIB   DD DSN=&CICSLIB,DISP=SHR   CICS APPLICATION STUBS
```

LCCPCXXL

This cataloged procedure invokes the CICS translator, the C++ translator, the compiler, COOL, and the linkage editor in one step. Input to this step is provided by the CCP.SYSIN DD statement and the link-edited load module is placed in the data set referred to on the LKED.SYSLMOD DD statement, as follows:

```
//PCL        EXEC LCCPCXXL,PARM.CCP='CICS options',
//         PARM.X='C++ options',PARM.LKED='COOL and Link Edit options'
//CCP.SYSIN    DD DSN=your.source.library(member),DISP=SHR
//LKED.SYSLMOD DD DSN=your.cics.load(member),DISP=SHR
```

LCCPCXXA

This cataloged procedure invokes the CICS translator, the C++ translator, the compiler, and the AR370 archive utility in a single step. Input to this step is provided by the CCP.SYSIN DD statement, and the object code generated by the compiler is placed in the archive referred to on the A.SYSARLIB DD statement, as follows:

```
//PCA        EXEC LCCPCXXA,PARM.CCP='CICS options',
//         PARM.X='C++ options',PARM.A='AR370 options'
//CCP.SYSIN    DD DSN=your.source.library(member),DISP=SHR
//A.SYSARLIB   DD DSN=your.ar.library,DISP=SHR
```

Preprocessing, Compiling, and Linking under CMS

CICS applications can also be developed under CMS. You can preprocess, compile, and link your application under CMS before shipping the resulting object module to an OS/390 or VSE system for final link-editing with the CICS Execution Interface stub routines. For complete details on compiling C programs under CMS, consult the SAS/C Compiler and Library User's Guide.

Before you run the translator, compiler, or COOL under CMS, verify that the transient library is available on an accessed minidisk, or that it is installed in a segment available to your virtual machine. Your SAS Software Representative for SAS/C software products can tell you if this has been done for you.

The Translator Input File

The input file is a C source file containing one or more EXEC CICS commands. The file may be in any format acceptable to the compiler. Thus, it can contain either variable-length or fixed-length records, with a logical record length less than or equal to 1024 characters.

The input file may have sequence numbers. The translator checks the first record in the source file to determine if the source file has sequence numbers. If the source file has variable-length records, the translator inspects columns 1 through 8; otherwise, it inspects the last eight columns. If the translator finds a sequence number in the first record, it ignores the corresponding columns of all subsequent records.

Note: The compiler will not accept an input file containing lines longer than 1024 characters. △

The Translator Output File

The output of the translator is subsequently used as input to the compiler. If the OUTSEQ option is used and the input data set does not have sequence numbers, the LRECL of the output file will be eight characters longer than the input file.

The Listing File

A listing file is created unless the NOPRINT option is used. It will have fixed-length records and a logical record length of 121. The listing contains a list of the LCCCP options in effect (unless the NOOPTIONS option is used), a source listing (unless the NOSOURCE option is used), and a cross-reference listing (unless the NOXREF option is used). Diagnostic messages are also written to the listing. If the EXPAND option is used, the translation of each CICS command is shown following the command in the source listing.

Terminal Output

LCCCP prints all diagnostic messages to the terminal unless the NOTERM option is in effect.

The LCCCP EXEC

The LCCCP EXEC invokes the translator in CMS. The format of the LCCCP EXEC follows:

LCCCP *filename* (*filetype* (*filemode*)) [OUT(*out-fileid*)] [PR(*print-fileid*)] ((*options*( )))

where

filename
is the filename of the input file.

filetype
is the filetype of the input file. If *filetype* is not specified, CCP is the default.

filemode
is the filemode of the input file. If *filemode* is not specified, A is the default.

out-fileid
is the fileid of the output file. If this option is not specified, the default fileid is *filename* C A1. If the OUT( )option is used, the filetype and filemode of the output file may be omitted. The default filetype is C and the default filemode is A1.

print-fileid
is the fileid of the listing file. If this option is not specified, the default fileid is *filename* CCPLIST A1. If the PR( )option is used, the filetype and filemode of the listing file may be omitted. The default filetype is CCPLIST and the default filemode is A1.

options
are any of the LCCCP options described in "Specifying Translator Options" on page 13.

Compiling SAS/C Programs for CICS under CMS

When you invoke the compiler with the LC370 EXEC, specify as input the same filename you used for output in the preprocessing step. If you are not familiar with the SAS/C Compiler, refer to SAS/C Compiler and Library User's Guide for additional information.

Note: CICS requires that all programs be compiled so that they are re-entrant. You must specify the compiler options **rent** or **rentext** to cause the compiler to generate re-entrant code, because the compiler default for re-entrancy is **norent**. △

For example, to compile the file FTOC.C.A, you issue the following command:

```
LC370 ftoc (rent)
```

Linking SAS/C Programs for CICS under CMS

Regardless of the linking method you use, you must always

- include the CICS Execution Interface stub routines in each load module
- arrange for the stub to be the very first thing in the load module.

The CLISTs and cataloged procedures distributed by SAS Institute have been written to automate this process by using the following linkage-editor control statements in one form or another:

```
LIBRARY DFHLIB(DFHEAI,DFHEAIO)
ORDER   DFHEAI
```

The DDname DFHLIB points to the CICS load library that contains the execution interface stub routines.

When to Use COOL

You must use COOL to preprocess your object code if

- two or more compilations in the program are compiled using either of the compiler options (**rent** or **rentext**)
- the program initializes external variables in two or more re-entrant compilations
- you use the all-resident library
- you specify the **EXTNAME** option for more than one compilation.

See "Using CMS EXECs to Link" on page 53 and the SAS/C Compiler and Library User's Guide for more information on COOL.

Linking All-Resident Programs

The default name of the all-resident library is CICSARES TXTLIB. (Ask your SAS Software Representative for SAS/C software products for the name of the library at your site.) When linking an all-resident program, concatenate the all-resident library in front of any other autocall data sets, and include the object deck created by compiling a source file that includes **<resident.h>** and the appropriate macro definitions. See the SAS/C Compiler and Library User's Guide for more information on linking all-resident programs. The process of linking all-resident programs is automated through the use of keywords and parameters in the cataloged procedures, CLISTs, and EXECs.

Using CMS EXECs to Link

The COOL EXEC invokes the COOL object code translator. The format is as follows:

COOL [*filename1*[*filename2* . . .]][(*options*[)]]

where *filename1*, *filename2*, and so on are the names of the files that are the primary input to COOL. Each file should have a filetype of TEXT and contain either object code or COOL/linkage-editor control statements. If no filenames are specified, COOL prompts for the name of a primary input file. At the prompt, enter a filename. COOL continues to prompt until a null line is entered. See "COOL Options" on page 55 for a list of options.

Several different techniques can be used to run COOL on applications for CICS on CMS. Each technique is just a different way of specifying which text libraries COOL should use for autocall resolution.

The simplest technique is to specify the CICS option to COOL. The CICS option directs COOL to use the LC370CIC TXTLIB and the LC370BAS TXTLIB as autocall libraries. An example invocation would be:

```
COOL ftoc (CICS)
```

A second technique is to use the CMS GLOBAL TXTLIB command to explicitly specify which TXTLIBs are to be used by COOL, and not to specify any COOL options. For example

```
GLOBAL TXTLIB LC370CIC LC370BAS
COOL ftoc
```

If you have your own library of routines to be used for autocall, then you could issue a GLOBAL TXTLIB command that specifies your TXTLIB before invoking COOL with the CICS option. For example

```
GLOBAL TXTLIB yourlib
COOL ftoc (CICS)
```

Alternatively, you could specify all the libraries to be used for autocall:

```
GLOBAL TXTLIB yourlib LC370CIC LC370BAS
COOL ftoc
```

You can also use the CMS GLOBALV command to create a default list of TXTLIBs that need to be global when COOL is invoked. The COOL EXEC queries the TXTLIBS variable in the group LC370 to determine whether any default TXTLIBs have been specified. The EXEC retains the current GLOBAL TXTLIBs before issuing a new GLOBAL command to be used during the invocation of COOL. It then restores the saved global TXTLIBs after COOL has terminated.

For example, the following CMS GLOBALV command specifies that the *yourlib* TXTLIB be made global when COOL is invoked. Because the CICS option is specified, the LC370CIC and LC370BAS TXTLIBs are also made global.

```
GLOBALV SELECT LC370 SETL TXTLIBS yourlib
CLIN ftoc (CICS)
```

The COOL EXEC accepts the NOGLOBAL option. If this option is used, the EXEC does not query the GLOBALV variable TXTLIBS for the names of TXTLIBs that are to be global before COOL begins execution.

Creating C++ CICS Applications under CMS

You must execute three separate EXECs to create a C++ CICS application under CMS:

LCCCP	invokes the CICS translator.
LCXXC	invokes the C++ translator and compiler.
COOL	invokes the SAS/C prelinker.

Output from COOL must be ported to the target operating system for final link-editing.

The LCCCP EXEC

The LCCCP EXEC requires an input file with a filetype of CPP. An output file will be created with the same filename as the input file and a filetype of C unless the **OUT** option is used to specify an alternate fileid. The following example uses the **OUT** option to specify a fileid with the filetype (CXX) expected by the LCXX EXEC.

Assuming an input file named SAMPLE CCP A and an output file named SAMPLE CXX A, the LCCCP EXEC can be invoked as follows:

```
LCCCP SAMPLE OUT(SAMPLE CXX A
```

The LCXX EXEC

The LCXX EXEC requires the filetype of the input file to be CXX. By default, an output file with the same filename as the input file and a filetype of TEXT will be created. Before executing the LCXX EXEC, the LC370 and LCXX370 MACLIBs must be made globally available.

Note: The **RENT** compiler option must always be specified for CICS applications.

Assuming an input fileid of SAMPLE CXX A and an output fileid of SAMPLE TEXT A, the LCXX EXEC can be invoked as follows: △

```
LCXX SAMPLE (RENT
```

Consult the SAS/C Cross-Platform Compiler and C++ Development System User's Guide for additional information on the LCXX EXEC.

At this point, you can do one of the following:

- use COOL to prelink the SAMPLE TEXT A file on CMS and then port it to the target operating system for final link-editing

□ port to the target operating system and then prelink (using COOL) and link-edit.

To prelink the output file on CMS, enter the following command:

```
COOL SAMPLE (CXX CICS
```

An output file named COOL370 TEXT A is created and ready for porting to the target operating system. You must use the **CXX** and **CICS** options to add the required libraries to COOL's autocall list.

If the target operating system is OS/390, you can invoke the LCCCXXL cataloged procedure with the **NOCOOL** option.

Note: You must always link-edit the application with the **RENT** option. Here's an example that invokes the LCCCXXL cataloged procedure: △

```
//LINK  EXEC LCCCXXL,PARM.LKED='NOCOOL,RENT'
//LKED.SYSLMOD DD DSN=your.cics.load(member),DISP=SHR
//*
//* Note that input must be provided via the SYSLIN DD statement
//* instead of the SYSIN DD statement when using the NOCOOL option.
//LKED.SYSLIN   DD DSN=your.object.library(member),DISP=SHR
```

Alternatively, you can directly invoke the linkage editor program, IEWL:

```
//LINK  EXEC PGM=IEWL,PARM='RENT'
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//SYSLMOD  DD DSN=your.cics.load(member),DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,50))
//DFHLIB   DD DSN=your.cicsrel.SDFHLOAD,DISP=SHR    Command-level stubs
//OBJIN    DD DSN=your.object.library(member),DISP=SHR
//SYSLIN   DD *
  LIBRARY DFHLIB(DFHEAI,DFHEAI0)
  ORDER DFHEAI
  ENTRY MAIN
  INCLUDE OBJIN(member)
  NAME SAMPLE(R)
/*
//
```

If you choose to port the application to OS/390 for prelinking (using COOL) and link-editing, use the LCCCXXL cataloged procedure, as described in "Creating C++ CICS Applications under OS/390 Batch" on page 47.

COOL Options

Table 5.1 on page 56 lists the options available for the COOL utility and the systems to which these options apply. A description of each option follows the table.

Table 5.1 COOL Options

Option	TSO	CMS	OS/390 Batch
ALLRESIDENT	X	X	
AUTO		X	
CICS	X	X	
CICSVSE	X	X	
CXX	X	X	
ENTRY	X		
ENXREF	X	X	X
EXTNAME	X	X	X
GLOBAL		X	
LIB	X		
LKED		X	
LOAD	X		X
LOADLIB	X		
PREM	X	X	X
PRINT		X	
SPE	X	X	
TERM	X	X	X
UPPER	X	X	X
WARN	X	X	X

ALLRESIDENT
specifies use of the all-resident library. Under CMS, this option issues GLOBAL TXTLIB commands for LCARES, LC370BAS, and LC370STD TXTLIBs. Under TSO, the following keyword, when used with either the **CICS** or **CICSVSE** keywords, names SASC.CICS.ARESOBJ, SASC.CICSOBJ, and SASC.BASEOBJ to be used automatically as call libraries:

```
ALLRESIDENT
```

This option should be specified when linking all-resident programs. See also the **CICS** and **CICSVSE** options.

AUTO
specifies that COOL should resolve external references by searching for object files whose filenames match the external reference. This is the default. **NOAUTO** suppresses resolution of external references by object files. The **AUTO** option is similar to the AUTO option of the CMS LOAD command. When **AUTO** is in effect, COOL attempts to resolve external references by searching for files named *ref* TEXT, where *ref* is the name of the external reference. If no TEXT file with that

name can be found, COOL attempts to resolve the reference from the GLOBAL TXTLIBs.

CICS
under CMS, makes the required TEXT libraries global for linking programs that execute in CICS. Usually, the LC370BAS and LC370CIC TXTLIBs are global. If you specify this option with the **ALLRESIDENT** option, the CICSARES, LC370CIC, and LC370BAS TXTLIBs are global. If you specify this option with the **SPE** option, the LC370BAS and LC370SPC TXTLIBs are global.

Under TSO, by default, this option specifies that SASC.BASEOBJ and SASC.CICSOBJ are to be used automatically as call libraries. If you specify this option with the **ALLRESIDENT** option, the SASC.CICS.ARESOBJ data set is also used as an autocall library. If you specify this option with the **SPE** option, the SASC.CICS.SPEOBJ and SASC.BASEOBJ data sets are used automatically as call libraries.

CICSVSE
under CMS, makes the required TEXT libraries global for linking programs that execute in CICS under VSE. Usually, the LC370BAS and LC370VSE TXTLIBs are global. If you specify this option with the **ALLRESIDENT** option, the CICSARES, LC370VSE, and LC370BAS TXTLIBs are global. If you specify this option with the **SPE** option, the LC370BAS and LC370SPC TXTLIBs are global.

Under TSO, by default, this option specifies that SASC.BASEOBJ and SASC.VSEOBJ are to be used automatically as call libraries. If you specify this option with the **ALLRESIDENT** option, the SASC.CICS.ARESOBJ data set is also used as an autocall library. If you specify this option with the **SPE** option, the SASC.CICS.SPEOBJ and SASC.BASEOBJ data sets are used automatically as call libraries.

CXX
specifies that the object code being linked is produced by compiling output from the C++ translator. This option must be used when linking C++ translator output. **NOCXX** specifies that the object code being linked is not produced by compiling output from the C++ translator. **NOCXX** is the default.

ENTRY
under TSO, identifies the program's entry point, or allows the linkage editor to determine the entry point when written in the following format:

ENTRY (*name*)

The **ENTRY** keyword can be specified in the following ways:

ENTRY(MAIN)
for a program containing a **main** function. The actual entry point is MAIN.

ENTRY(DYN)
for a re-entrant, dynamically loaded (via **loadm**) module. The actual entry point is #DYNAMN.

ENTRY(CSPE)
for a CICS SPE application with initial function **cicsmain**. The actual entry point is #CICSEP.

ENTRY(NONE)
to allow the linkage editor to select the entry point itself. Use of **ENTRY(NONE)** is recommended only if a linkage-editor **ENTRY** statement is present in one of the input files.

If **ENTRY** is not specified, **ENTRY(MAIN)** is assumed, unless **SPE** is specified; in that case, **ENTRY(NONE)** is assumed.

ENXREF
controls the production of one or more of the three cross-references generated by COOL in a table that follows all other COOL output. These three cross-references are **SNAME**, **CID**, and **LINKID**. The cross-reference **SNAME** is in alphabetic order by the **SNAME** that uniquely identifies an object file. **CID** displays the extended names in alphabetic order by the C identifier. **LINKID** displays the extended names in alphabetic order by a linkid that COOL assigns. **NOENXREF** suppresses the production of all extended names cross-references.

Under TSO, the **ENXREF** option takes the following form:

```
ENXREF('NOSNAME,NOCID,NOLINKID')
```

For example, the following option suppresses the **SNAME** cross-reference:

```
ENXREF('NOSNAME')
```

Under CMS, the **ENXREF** option takes the following form: 9pt

```
ENXREF <NOSNAME> <NOCID> <NOLINKID>
```

For example, the following option suppresses the **SNAME** cross-reference:

```
ENXREF NOSNAME
```

Under OS/390 batch, the **ENXREF** option takes the following form:

```
ENXREF(NOSNAME,NOCID,NOLINKID)
```

For example, the following option suppresses the **SNAME** cross-reference:

```
ENXREF(NOSNAME)
```

EXTNAME
specifies that COOL is to process extended names. This is the default. **NOEXTNAME** specifies that COOL will not process extended names.

The SAS/C Compiler provides extended names support that enables compiler processing of extended names of up to 64K in length. An *extended name* is any name that identifies an external variable, or that identifies an external or static function and fits either of the following criteria:

- It is more than eight characters long.
- It is eight characters or fewer in length, contains uppercase alphabetic characters, and is not the name of an **_asm** or HLL (for example, **_pascal**) function.

Note: If you specify the **EXTNAME** option, be sure to include the appropriate header files for library functions that you use. Some library functions, such as **localtime** and **setlocale**, are more than eight characters long and, therefore, fit the criteria for extended names. The library header files for these functions all contain **#pragma map** statements that change the function names to names that are not extended. △

If you do not include the appropriate library header file for a library function, the compiler creates unpredictable external symbols that cannot be resolved from the standard library. For more information on **#pragma map**, refer to Chapter 2, "Source Code Conventions," in the SAS/C Compiler and Library User's Guide.

GLOBAL
specifies that the COOL EXEC should query the GLOBALV variable TXTLIBS in the group LC370 for the name or names of TXTLIBs that are to be global before COOL begins execution. This is the default. **NOGLOBAL** suppresses automatic

query of the GLOBALV variable TXTLIBS: the EXEC does not issue a GLOBAL TXTLIB command based on the GLOBALV variable.

LIB
specifies the data set name of an autocall object library containing functions that are to be linked automatically into the program if referenced. (Load module libraries cannot be used.) Here is the form of the keyword:

```
LIB (dsname)
```

If the library belongs to another user, the fully qualified name of the data set must be given, and the name must be preceded and followed by three apostrophes. No final qualifier is assumed for a **LIB** data set.

LKED
specifies that the COOL EXEC is to issue an **LKED** command for COOL370 TEXT using the **LKED** options specified. The **LKED** option must follow any use of any other option on the command line. The **LKED** option causes the COOL EXEC to issue the following CMS command after COOL has created the COOL370 TEXT file:

```
LKED COOL370 (options)
```

where *options* are any **LKED** command options specified following the **LKED** keyword.

LOAD
names the data set into which the linkage editor stores the output load module. Typically, this will be the name of an application load library appearing in the CICS DFHRPL DDname concatenation of libraries. Here is the form of the keyword:

```
LOAD(dsname)
```

This keyword should specify a partitioned data set member. If the data set belongs to another user, the fully qualified name of the data set must be given, and the name must be preceded and followed by three apostrophes. If the data set name is not specified within three apostrophes, it is assumed to be a data set name with a final qualifier of **LOAD**. Additional considerations follow:

- If the **LOAD** keyword is not used, the load module data set is determined by replacing the final **OBJ** qualifier in the object data set name with **LOAD**.
- If a member name is specified for the object data set, the same member name is assumed for the load module; if the object data set is a sequential data set, the member name TEMPNAME is assumed for the load module name.
- If the object data set name is specified in apostrophes, the terminal user is prompted to enter the name of the **LOAD** data set.

LOADLIB
specifies the data set name of an autocall load library containing functions that are to be linked automatically into the program if referenced. Here is the form of the keyword:

```
LOADLIB(dsname)
```

If the library belongs to another user, the fully qualified name of the data set must be given, and the name must be preceded and followed by three apostrophes. Functions in the **LOADLIB** data set are resolved by the linkage editor, not by COOL. COOL diagnoses these functions as unresolved. No final qualifier is assumed for a **LOADLIB** data set. You must use **LOADLIB**, rather than **LIB**, to reference libraries that are associated with IBM products such as ISPF and GDDM because those libraries are stored in load module format.

PREM
specifies that COOL is to remove pseudoregisters from the output object module. COOL creates a pseudoregister map. **NOPREM** specifies that COOL is not to remove pseudoregisters from the output object module. **PREM** is the default under CMS; **NOPREM** is the default under OS/390. The **PREM** option is rarely used in TSO or under OS/390. However, if you are linking programs for CICS/VSE, you must specify this option.

PRINT
indicates the destination for COOL and linkage editor output listings. The following keyword indicates that the COOL and linkage editor output listings should be printed at the terminal:

```
PRINT(*)
```

The following keyword specifies that the COOL and linkage editor listings should be stored in the named data set:

```
PRINT (dsname)
```

This data set must be sequential; a partitioned data set member is not allowed. If the data set belongs to another user, the fully qualified name of the data set must be given, and the name must be preceded and followed by three apostrophes. If the data set name is not specified within three apostrophes, it is assumed to be a data set name with a final qualifier of LINKLIST.

The following keyword specifies that no linkage editor or COOL listing is to be produced:

```
NOPRINT
```

If you use the **NOPRINT** operand with CLK370, COOL and linkage-editor output (except for diagnostic messages) is suppressed.

If neither **PRINT** nor **NOPRINT** is used, the default is **NOPRINT**.

SPE
When **SPE** is used while also specifying the CICS or CICSVSE keywords, the CICS SPE libraries are used to link the application. Under CMS, the GLOBAL TXTLIB command is issued for the LC370SPC and LC370BAS TXTLIBs. Under TSO, the GLOBAL TXTLIB command causes the SASC.CICS.SPEOBJ and SASC.BASEOBJ data sets to be used automatically as call libraries.

TERM
specifies that COOL error messages be written to **stderr** (SYSTERM) in addition to **stdout**. **NOTERM** suppresses the error message listing to **stderr**. This is the default.

UPPER
produces all output messages in uppercase.

WARN
specifies that warning messages (which are associated with RC=4) be issued. This is the default. **NOWARN** suppresses warning messages.

COOL Messages and CICS Applications

When you use COOL to link a CICS application, you receive COOL messages noting the missing CICS Execution Interface stubs. Under CMS, you receive the following messages:

```
LSCL102  Warning: Can't open file during autocall: %TXTLIB(DFHEI1)
LSCL102  Warning: Can't open file during autocall: %TXTLIB(DFHEAI)
LSCL102  Warning: Can't open file during autocall: %TXTLIB(DFHEAI0)
```

Under OS/390, you receive similar messages that refer to members of the SYSLIB library (instead of the CMS TXTLIB library). These messages are expected; the output object file from COOL must subsequently be linked with the CICS Execution Interface stubs. This subsequent link must be performed under OS/390, which means that if you are running under CMS, your output file (COOL370 TEXT) must be transmitted to an OS/390 system before it can be linked with the CICS Execution Interface stubs.

Note: Because final link-editing with the CICS execution interface stubs must be performed on the target operating system, if your application is targeted for a CICS/VSE system, refer to the next section. △

Linking for VSE

CICS application programs that are intended to be run under CICS and VSE may be developed on CMS or OS/390 systems. Generally, the technique is to preprocess, compile, and link your application under CMS or OS/390 before shipping the resulting object module to a VSE system for final link-editing with the CICS Execution Interface stub routines. Your applications are preprocessed and compiled in the normal manner under CMS or OS/390.

Note: The VSE linkage editor does not support pseudoregisters. Therefore, the COOL **PREM** option should be used to remove pseudoregisters from the object code. This change does not affect program execution or re-entrancy in any way. △

Linking CICS/VSE Applications under CMS

The CICS/VSE resident library is contained in the LC370VSE TXTLIB and LC370BAS TXTLIB libraries. The simplest method of linking applications for CICS and VSE is to specify the **CICSVSE** option to COOL. Here is an example:

```
COOL ftoc (CICSVSE)
```

You can also use any of the alternate COOL processing techniques described earlier in "Using CMS EXECs to Link" on page 53. For example

```
GLOBAL TXTLIB LC370VSE LC370BAS
COOL ftoc

GLOBAL TXTLIB yourlib
COOL ftoc (CICSVSE)

GLOBALV SELECT LC370 SETL TXTLIBS yourlib
COOL ftoc (CICSVSE)
```

Note: Pseudoregister removal is the default option on CMS. △

The resulting object file, COOL370 TEXT, must be transmitted to the destination VSE system for final link-editing. You can ignore any LSCL102 warning messages that reference the CICS Execution Interface stub routines.

Linking CICS/VSE Applications under OS/390

Under OS/390, the CICS and VSE resident library is contained in SASC.VSEOBJ and SASC.BASEOBJ. You can use the cataloged procedure LCCCL to create an object file for subsequent link-editing on a VSE system. You must code the ENV=VSE keyword in the JCL you use to invoke the procedure, and you must specify the COOL option **PREM** to remove pseudoregisters from the resulting object file. An example invocation of the cataloged procedure follows:

```
//    EXEC LCCCL,ENV=VSE,PARM='PREM'
//SYSLIN   DD DISP=SHR,DSN=your.object.library(outmem)
//SYSIN    DD DISP=SHR,DSN=your.object.library(inputobj)
```

The SYSLIN DD statement points to the data set that contains the resulting object file. The SYSIN DD statement points to any input object files. You may ignore any COOL LSCL102 warning messages that reference the CICS Execution Interface stub routines.

Creating the VSE Phase

After you have linked your application on your development system, the resulting object file must be transmitted to the destination VSE system for final link-editing with the CICS Execution Interface stubs to create an executable phase. Example Code 5.7 on page 62 shows how this can be done.

Example Code 5.7 Sample VSE Linking JCL

```
// JOB
// LIBDEF *,...
 PHASE load_module_name,*
 INCLUDE DFHEAI
 INCLUDE DFHEAIO
 INCLUDE
        .
        . /* object code produced by COOL */
        .
/*
// EXEC LNKEDT
/*
```

Using the External CICS Interface

The CICS translator now supports the external CICS interface. By using this feature, a non CICS program running on OS/390 can call a CICS program running in a CICS region. This is sometimes referred to as a Distributed Program Link (DPL). You can use one of two programming interfaces to enable the external CICS interface:

- the EXCI CALL interface
- the EXEC CICS interface.

Consult the CICS/ESA External CICS Interface Manual for more information on the EXCI CALL interface.

The EXEC CICS Interface

You invoke the EXEC CICS interface by using new options in the **EXEC CICS LINK PROGRAM** command: **RETCODE(*data-area*)**, and **APPLID(*name*)**. These options are valid only when you specify the external call interface option.

When you specify the external call interface, the following options for the **EXEC CICS LINK PROGRAM** command are not valid:

INPUTMSG(*data-area*)
INPUTMSGLEN(*data-value*)
SYSID(*data-area*)

Running your CICS Application with the External CICS Interface

The following three steps are required to run your CICS application with the external CICS interface:

1. The CICS command translator must translate your program.
2. You must compile the program with SAS/C.
3. You must link-edit the program on OS/390.

Note: Your program must be linked with the standard resident library, STDOBJ, rather than the CICS library. △

Translating, Compiling, and Linking under OS/390

Use the **EXCI** option to specify the external call interface on OS/390. To resolve the external CICS interface stub when linking, you must specify the CICS41.SDFHEXCI library. Ask your CICS system administrator for the exact name of the SDFHEXCI data set at your site.

Note: You can use this feature of the CICS command translator in the UNIX environment. See SAS/C Cross-Platform Compiler and C++ Development System User's Guide for details. △

Using TSO

To translate the program, run the LCCCP CLIST:

```
LCCCP 'your.ccpinput.ccp(member)' OUTPUT('''ccp.output.c(member)''')
       EXCI
```

Next, compile the program:

```
LC370 'ccp.output.c(member)' OBJECT('''ccp.output.load(member)''')
       RENT
```

Link the program:

```
COOL 'ccp.output.obj(member)' LOADLIB('''CICS41.SDFHEXCI''')
      LOAD('''ccp.output.load(member)''') LET
```

To execute the program under TSO, you must allocate the CICS41.SDFHEXCI library in one of the following ways:

Example Code 5.8 Method 1

```
ALLOC FILE(ISPLLIB) DATASET('CICS41.SDFHEXCI') SHR
```

To run the program using Method 1, type the following:

```
CALL 'ccp.output.load(member)'
```

Example Code 5.9 Method 2

```
ALLOC FILE(CPLIB) DATASET('ccp.output.load' 'CICS41.SDFHEXCI') SHR
```

To run the program using Method 2, type the following:

```
member
```

Using OS/390 Batch

You can use one of three procedures (LCCCP, LCCCPC, or LCCCPCL) to invoke the CICS translator. If you use either LCCCP or LCCCPC when translating for the external CICS interface, you can invoke the linker separately and specify the correct libraries.

For example, to translate and compile the program, run the LCCCPC procedure:

```
//           EXEC LCCCPC,PARM.CCP='EXCI'
//SYSPUNCH DD DISP=SHR,DSN=your.object.library(member)
//SYSIN    DD DISP=SHR,DSN=your.source.library(member)
```

Then link and execute the program:

```
//STEP1    EXEC LC370LRG
//LKED.SYSLMOD DD DISP=SHR,DSN=your.load.lib(member)
//LKED.SYSIN   DD DISP=SHR,DSN=your.object.lib(member)
//LKED.SYSLDLIB DD DISP=SHR,DSN=CICS41.SDFHEXCI
//GO.STEPLIB DD
//         DD DISP=SHR,DSNAME=CICS41.SDFHEXCI
```

Translating, Compiling, and Linking under CMS

Use the **EXCI** option to specify the external call interface on CMS. First, translate the program using the LCCCP EXEC:

```
LCCCP filename (filetype (filemode)) (EXCI
```

Next, compile the program:

```
LC370 filename (filetype (filemode)) (RENT
```

Finally, the object must be moved to OS/390 to execute the link step. This step is illustrated in "Translating, Compiling, and Linking under OS/390" on page 63.

Using SQL and C

As noted in Chapter 8, "Handling Files," on page 77, the SAS/C Compiler supports development of C language application programs that contain Structured Query Language (SQL) statements. Such programs must first be precompiled with an SQL preprocessor (not provided by SAS Institute) that translates the EXEC SQL statements

to valid C code. Example Code 5.10 on page 65 provides a sample of JCL for such applications.

Example Code 5.10 Sample JCL for SQL

```
//JOB CARD INFO
//JOBLIB  DD  DISP=SHR,DSN=db2.library.DSNLOAD
//*******************************************************************
//*        STEP 1 : PRECOMPILE                                      *
//*******************************************************************
//PC       EXEC PGM=DSNHPC,
//  PARM='HOST(C),STDSQL(NO),SOURCE,XREF,MARGINS(1,72)'
//DBRMLIB  DD  DSN=db2.dataset.DBRMLIB.DATA(member),DISP=SHR
//STEPLIB  DD  DSN=db2.library.DSNEXIT,DISP=SHR
//         DD  DSN=db2.library.DSNLOAD,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSTERM  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSCIN   DD  DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA
//             SPACE=(800,(500,500))
//SYSLIB   DD  DSN=db2.library.SRCLIB.DATA,DISP=SHR
//SYSUT1   DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSUT2   DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
//SYSIN    DD  DSN=your.sample.program(member),DISP=SHR
//*******************************************************************
//*        STEP 2 : TRANSLATE, COMPILE AND LINK                     *
//*******************************************************************
//TCL  EXEC LCCCPCL
//CCP.SYSIN    DD DSN=&&DSNHOUT,DISP=(OLD,DELETE)
//LKED.SYSLIB  DD
//             DD
//             DD
//             DD DSN=db2.library.DSNLOAD,DISP=SHR
//LKED.SYSLMOD DD DSN=your.cics.application.loadlib(member),DISP=SHR
//LKED.SYSIN   DD *
     INCLUDE SYSLIB(DSNCLI)
//*******************************************************************
//*        STEP 3 : BIND                                            *
//*******************************************************************
//BIND     EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB   DD  DISP=SHR,DSN=db2.dataset.DBRMLIB.DATA
//SYSTSPRT  DD  SYSOUT=*
//SYSTERM   DD  SYSOUT=*
//SYSPRINT  DD  SYSOUT=*
//CEEDUMP   DD  SYSOUT=*
//SYSUDUMP  DD  SYSOUT=*
//SYSOUT    DD  SYSOUT=*
//REPORT    DD  SYSOUT=*
//SYSIN     DD  *
//SYSTSIN   DD  *
 DSN SYSTEM(yoursystem)
 BIND PLAN(yourplan) MEMBER(member) ACT(REP) ISOLATION(CS)
 END
//
```

Diagnostics

The translator produces diagnostic messages with four levels of severity. Diagnostic messages usually change the return code set by LCCCP0. The levels of severity are explained here:

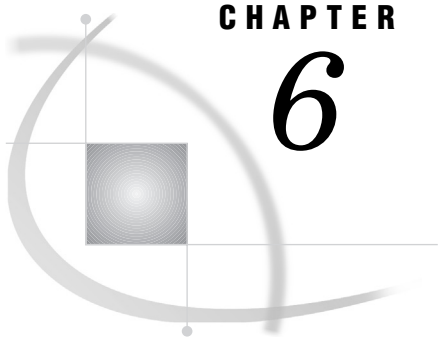
Note
: provides information about expected behavior. The return code is not changed.

Warning
: provides information about conditions that will not prevent translation from completing successfully. The return code is set to 4.

Error
: provides information about conditions that will prevent translation from continuing. If an error occurs before translation begins (such as a file I/O error), LCCCP will terminate immediately. If an error occurs after translation has begun, LCCCP will stop translating but will check all the remaining commands in the input file for correct syntax. The return code is set to 8.

Severe Error
: provides information about conditions that will cause immediate termination. The return code is set to 12.

A complete listing and explanation of translator diagnostic messages can be found in SAS/C Software Diagnostic Messages.



CHAPTER 6

Running and Debugging SAS/C Programs in the CICS Environment

Introduction **67**
Running SAS/C Programs under CICS **67**
Using Run-Time Options **68**
Specifying Run-Time Options Using External Variables **68**
Specifying Run-Time Arguments with $MAINO **69**
Environment Variable Support **69**
Suggestions for Efficiency **69**
Debugging Your SAS/C CICS Application **69**
Abend Codes **70**

Introduction

This chapter describes what you need to know to run your SAS/C program in the CICS environment. The SAS/C Library includes the function **`iscics()`**, which returns an indication to the program about whether the program is running in a CICS environment. Consult the SAS/C Library Reference, Volume 1 for more details.

Running SAS/C Programs under CICS

Before you can run SAS/C applications under CICS, you must perform the following steps. Step 1 involves the SAS/C Command Language Translator and Compiler; the remaining steps are required by CICS, regardless of the application language.

1. Before you can run your program under CICS, you must have successfully translated, compiled, and linked your program using the SAS/C Translator and Compiler. Chapter 5, “Preprocessing, Compiling, and Linking,” on page 35 explains this process in detail.
2. The load module resulting from Step 1 must be stored in a load library that is contained in the CICS load module library concatenation (DDname DFHRPL). Ask your CICS system administrator for information on how to store the load library at your site.
3. To identify the program to CICS, you must define the program name (load module name) in the processing program table (PPT). Remember that SAS/C load modules (programs) should be specified as assembler language programs, not as C programs. You must also define in the PPT (with the appropriate language specified) any other programs that will be dynamically loaded by the SAS/C application program. You must also define any basic mapping support (BMS) maps.

4. Define the initiating CICS transaction code in the program control table (PCT). You must specify on the PCT entry the name of the module to be loaded and executed when the transaction is initiated.
5. If your program reads or writes to any files, you must define those files in the file control table (FCT).
6. Just as you defined FCT entries in the previous step, you must define any extrapartition or intrapartition destinations in the destination control table (DCT).
7. You must define the BMS maps (if any) used by your application. For CICS, this usually involves the following steps:
   a. Code the maps with macros.
   b. Assemble the maps to create a physical map.
   c. Create a logical map.

Note: In addition to these steps, for C programs on pre-ESA versions of CICS, you must process the symbolic assembly output using the DSECT2C utility. Chapter 7, "Terminal Control and Basic Mapping Support," on page 71 describes how to use this utility. △

After you have defined everything, you can initiate the program as you would any other CICS application. The usual method is to enter the transaction identifier in the upper left corner of the screen, and then to press ENTER.

Using Run-Time Options

Because CICS applications communicate with the user via full-screen panels or displays, there is no command line in the CICS environment. This environmental restriction prohibits the normal specification of run-time library options and environment variables as well as any application parameters. However, using C, you can still use run-time options by specifying them using external variables, by using the **$MAINO** entry point, or by using environment variables.

Specifying Run-Time Options Using External Variables

You can code run-time options in your source code by using any of the following external variables:

- **__options**
- **__negopts**
- **__linkage**
- **__stack**
- **__heap**
- **__mneed**

Initialize the integer variable **__options** to specify general run-time options in your program. Use the integer variable **__negopts** to reset or turn off one or more options.

Use the character variable **__linkage** to specify linkage options that control which prolog and epilog code should be executed with your program. Specify initial stack or heap allocation, or both, by using the options **__stack**, **__heap**, and **__mneed**.

Note: Pre-ESA versions of CICS do not allow an EXEC CICS GETMAIN request for greater than 65,504 bytes below the 16-megabyte line. In such an environment, for

AMODE 24 programs, specifying an initial stack allocation size equal to or greater than 60K using the option **__stack** will result in a CICS ASCR abend. In other words, you can have more than 60K of stack allocated, but the initial stack allocation must be smaller than 60K. △

Besides these external variables, all run-time arguments available with the SAS/C compiler can be used in your application program under CICS. For more information on using run-time arguments in your SAS/C programs, see Chapter 8, "Run-Time Argument Processing" in the SAS/C Compiler and Library User's Guide.

Specifying Run-Time Arguments with $MAINO

Run-time arguments can also be passed from a calling program if you use the **$MAINO** entry point. The **$MAINO** entry point processes library run-time options and parameters. Consult Chapter 11, "Communication with Assembler Programs," in the SAS/C Compiler and Library User's Guide for details on how to communicate using **$MAINO** and other C entry points.

Environment Variable Support

By using SAS/C, it is possible to specify run-time options under CICS by using environment variables. Consult Chapter 4, "Environment Variables," and the function descriptions for **getenv** and **putenv** in the SAS/C Library Reference, Volume 1 for details on this support.

Suggestions for Efficiency

In general, the following guidelines will help your SAS/C program run more efficiently:

- Run with the **=usage** run-time option to help you calculate heap and stack size allocations during testing.
- Evaluate selecting faster prologs.
- Use inlined functions wherever possible.

You may find it helpful to review relevant sections in the SAS/C Compiler and Library User's Guide. You may also want to review sections on optimizing your SAS/C program. For CICS application design considerations, additional information is available in IBM's CICS Performance Guide.

Debugging Your SAS/C CICS Application

SAS/C programs fully support the command-level Execution Diagnostic Facility (EDF). EDF views a SAS/C application program as an assembler language program. When debugging your application using EDF, you may see the message **R13 does not address DFHEISTG**. For SAS/C CICS applications, you can ignore this message.

Note: The run-time library may issue CICS commands on behalf of an application, but because the library uses the **NOEDF** option, these commands will never be displayed. △

For details on debugging CICS applications, consult the following documents:

- *IBM CICS-Supplied Transactions*
- *IBM Application Programmer's Reference Manual (Command Level)*
- *SAS/C Debugger User's Guide and Reference*
- *SAS/C C++ Development System User's Guide*.

Abend Codes

If your program does not check for exceptional conditions (either via the RESP option in CICS commands, by establishing HANDLE CONDITION error handlers, or by synchronous signal handlers), CICS will either attempt to continue execution, or it will abnormally terminate execution of the program. If the execution is abnormally terminated, CICS will terminate the task with the abend code associated with the exceptional condition. However, if SAS/C library handling is in effect, the library will attempt to issue a diagnostic and a traceback, if appropriate.

SAS/C traceback information is directed to the **stderr** function. By default, this is the CICS transient data destination SASE. Output to **stderr** can be redirected to other files. See Chapter 8, "Handling Files," on page 77 for additional information.

The following abend codes are specific to SAS/C processing under CICS:

```
1229  No storage available for unconditional
      GETMAIN request.

1239  Unexpected internal error during abnormal
      termination processing.

12xx  Unable to locate
      TWA (Transaction Work Area) for
      lndep-compiled applications.
```

A complete listing of abend codes can be found in SAS/C Software Diagnostic Messages.

CHAPTER 7

Terminal Control and Basic Mapping Support

Introduction **71**
Terminal Control **71**
Basic Mapping Support **72**
Using LANG=C **72**
Using DSECT2C **73**
DSECT2C-generated Variable and Macro Names **74**
Example of Using BMS with C **75**
BMS Input/Output Operations **76**

Introduction

Input/output operations in the Customer Information Control System (CICS) environment can be provided in two ways: terminal control, and basic mapping support (BMS). Terminal control provides commands for unformatted communication. With BMS, you can define screens and communication with the terminal in a formatted manner. For more information on terminal control and BMS, consult the IBM documentation appropriate for your site.

Terminal Control

Your application program communicates with the terminal via the CICS terminal control program (TCP) using information defined in the terminal control table (TCT). Terminal control implies use of unformatted screens, as described in Chapter 4, "Tutorial: Creating a Simple Transaction," on page 29. The following commands are commonly used:

- HANDLE AID
- ISSUE COPY
- ISSUE PRINT
- RECEIVE
- SEND

Of these commands, only HANDLE AID has SAS/C product dependencies. HANDLE AID is implemented by specifying a function name for the error handler, rather than specifying a label, as in this example:

```
EXEC CICS HANDLE AID option(function-name) ... ;
```

See "Handle Condition and Handle AID" on page 24 for more information on this command.

Basic Mapping Support

BMS is provided by CICS to help you define and format screens. Using BMS frees the applications programmer from many lower-level details of screen definition, particularly details associated with device dependence and I/O operations.

When you define a screen using BMS, the screen is called a *map*. A collection of screens is called a *mapset*. Both physical (used primarily by CICS) and logical or symbolic maps (used primarily by the application program) must be created.

To create the original map definition, CICS provides assembler-language macros for use under BMS. For more information about defining maps and using these macros, consult the IBM CICS Application Programmer's Reference appropriate for your site.

Commonly used macros include the following:

DFHMDF
: defines a field in a map and its characteristics.

DFHMDI
: defines a map and its characteristics within a mapset.

DFHMSD
: defines a mapset and its characteristics and ends a definition.

After you finish creating a map, the map definition must be assembled and link-edited. This is done once for the physical map, and once for the symbolic map. The final destination for the physical map is the CICS load module library; the symbolic map goes into the header file library. Your site probably provides cataloged procedures that simplify this process.

You can use either of the following techniques to generate symbolic maps that are suitable for use in your C application:

- You can specify **`LANG=C`** in the DFHMSD macro.
- You can specify **`LANG=ASM`** and convert the generated DSECT to a C structure using the DSECT2C utility. For systems prior to CICS/ESA, this technique is the only one available.

Using LANG=C

If you use symbolic maps generated by specifying **`LANG=C`** in the DFHMSD macro, you must either specify the CBMSMAPS option to the CICS translator, or you must use the FROM option of the SEND MAP command and the INTO option of the RECEIVE MAP command. This example of code for BMS macros illustrates the use of the LANG=C option:

```
M0      DFHMSD TYPE=MAP,
               LANG=C,
               MAPATTS=(COLOR,HILIGHT),
               EXTATT=YES,
               MODE=INOUT,
               CTRL=FREEKB,
               STORAGE=AUTO,
               TIOAPFX=YES
M01     DFHMDI SIZE=(3,80),LINE=1
*
        DFHMDF ...
*
M02     DFHMDI SIZE=(3,80),LINE=4
```

```
*
        DFHMDF ...
*
        DFHMSD TYPE=FINAL
        END
```

Here's an example of C code that uses the CBMSMAPS option and a symbolic map named "mapmm0":

```
#pragma options xopts(cbmsmaps)
#include "mapm0.h"


void main()
{
   EXEC CICS SEND   MAP("m01")
                    MAPSET("m0")
                    ERASE
                    ;
}
```

Here's what the equivalent command would look like if you did not specify the CBMSMAPS option:

```
#pragma options xopts(cbmsmaps)
#include "mapm0.h"


void main()
{
   EXEC CICS SEND   MAP("m01")
                    MAPSET("m0")
                    ERASE
                    FROM (M01.MM01o)
                    ;
}
```

Using DSECT2C

When using C as the application programming language, create physical and logical maps for your mapset as described earlier. However, in addition to these steps, you must also process the assembler-language output listing for the symbolic assembly through the SAS/C utility, DSECT2C. This utility creates a C structure that must be included into the SAS/C application program source. It is recommended that you use **`#include`**, but you may also in-line the C structure into the source.

The DSECT2C utility, provided as part of the SAS/C product, converts assembler-language dummy sections, known as DSECTs, to equivalent C structure definitions. (For details on using this utility, refer to the SAS/C Compiler and Library User's Guide.) The SAS/C translator expects DSECT2C to generate the structure used to represent a map.

The **`-d`** option for the DSECT2C utility facilitates BMS map generation. By default, DSECT2C generates a structure definition only, as shown in the following example:

```
struct XYZ {
.
.
```

```
.
};
```

When you specify the **-d** option, DSECT2C adds an identifier to the definition, thereby declaring a variable of the structure type. The identifier name is the structure tag in lowercase, as shown here:

```
struct XYZ {
.
.
.
} xyz;
```

When you use the SEND MAP or RECEIVE MAP commands with string literal map names, the FROM/INTO options for these commands are optional. The translator will generate a default FROM/INTO argument that is the map name in lowercase. For example, the following commands are equivalent:

```
SEND MAP("MAP1");

SEND MAP("MAP1") FROM(map1)...;
```

Therefore, a simple way to use BMS maps is to use **-d** with DSECT2C to generate the structure declaration in a header file. When you include the header file (outside of any function), you are, in effect, declaring an **extern** variable of the structure type. When you use SEND MAP or RECEIVE MAP and allow the FROM/INTO options to default, the map will be sent from, or received into, the external structure.

The translator also generates a default length for the SEND command. BMS map generation always assigns the map length to a symbol whose name is the map name (in uppercase) followed by an E. In C, this symbol is a macro name. For example, if the length of MAP1 is 28, then the DSECT2C-created structure will include the following statement:

```
#define MAP1E 28
```

The translator uses this symbol as the default LENGTH value. Therefore, the following commands are equivalent:

```
SEND MAP("MAP1");

SEND MAP("MAP1") FROM(map1) LENGTH(MAP1E);
```

DSECT2C-generated Variable and Macro Names

The following characters, when used as the first character, are legal in a map name but not in a C variable name or a macro name: @, #, and $. For example, #MAP1 is a legal BMS map name, but #map1 is not valid as a C variable name. DSECT2C replaces these unacceptable characters with a digraph when it generates a variable name or a macro name. The character @ is replaced by A_, # with P_, and $ with D_. For example, the length of "MAP#1" will be defined via a macro named MAPP_1E, and the length of "$MAP1" will be defined via D_MAP1E.

Because the translator is aware of these changes, the correct defaults are generated. For example, the command **SEND MAP("MAP#1");** causes the translator to emit default values equivalent to the following:

```
SEND MAP("MAP#1") FROM(mapp_1) LENGTH(MAPP_1E);
```

Example of Using BMS with C

Example Code 7.1 on page 75 uses one of the maps specified for the sample application found in Appendix 1, "Examples," on page 99. Map SASCAGA is created in mapset MAPSETA. The BMS definition for the map follows the display.

Example Code 7.1 Sample Map

```
              +OPERATOR INSTRUCTIONS
               ---------------------

+OPERATOR INSTR - ENTER SMNU
+FILE INQUIRY   - ENTER SINQ AND NUMBER
+FILE BROWSE    - ENTER SBRW AND NUMBER
+FILE ADD       - ENTER SADD AND NUMBER
+FILE UPDATE    - ENTER SUPD AND NUMBER

+PRESS CLEAR TO EXIT
+ENTER TRANSACTION:+    +NUMBER+      +
```

The BMS map definition for this map is shown here:

```
TITLE 'FILEA - MAP FOR OPERATOR INSTRUCTIONS - SAS/C'
MAPSETA  DFHMSD TYPE=&SYSPARM,MODE=INOUT,CTRL=(FREEKB,FRSET),          *
               LANG=ASM,TIOAPFX=YES,EXTATT=MAPONLY,COLOR=BLUE
SASCAGA  DFHMDI SIZE=(12,40)
         DFHMDF POS=(1,10),LENGTH=21,INITIAL='OPERATOR INSTRUCTIONS',  *
               HILIGHT=UNDERLINE
         DFHMDF POS=(3,1),LENGTH=29,INITIAL='OPERATOR INSTR - ENTER*
               SMNU'
         DFHMDF POS=(4,1),LENGTH=38,INITIAL='FILE INQUIRY   - ENTER*
               SINQ AND NUMBER'
         DFHMDF POS=(5,1),LENGTH=38,INITIAL='FILE BROWSE    - ENTER*
               SBRW AND NUMBER'
         DFHMDF POS=(6,1),LENGTH=38,INITIAL='FILE ADD       - ENTER*
               SADD AND NUMBER'
         DFHMDF POS=(7,1),LENGTH=38,INITIAL='FILE UPDATE    - ENTER*
               SUPD AND NUMBER'
MSG      DFHMDF POS=(11,1),LENGTH=39,INITIAL='PRESS CLEAR TO EXIT'
         DFHMDF POS=(12,1),LENGTH=18,INITIAL='ENTER TRANSACTION:'
         DFHMDF POS=(12,20),LENGTH=4,ATTRB=IC,COLOR=GREEN,             *
               HILIGHT=REVERSE
         DFHMDF POS=(12,25),LENGTH=6,INITIAL='NUMBER'
KEY      DFHMDF POS=(12,32),LENGTH=6,ATTRB=NUM,COLOR=GREEN,            *
               HILIGHT=REVERSE
         DFHMDF POS=(12,39),LENGTH=1
         DFHMSD TYPE=FINAL
         END
```

After the map has been defined by using assembler-language macros, the map is assembled, and the DSECT2C utility is invoked to create the C header defining the map.

Under OS/390, the following JCL illustrates how a BMS map definition can be converted to a C structure:

```
//SASCAMA JOB ...(your job accounting info )...
//*
```

```
//*    EXECUTE THE IBM-SUPPLIED BMS CATALOGED PROCEDURE
//*    TO CREATE A LOGICAL MAP FOR USE BY CICS AND A
//*    SYMBOLIC MAP TO BE USED DURING DSECT2C PROCESSING
//* //  EXEC DFHMAPS,
//  DSCTLIB='your.symbolic.map.library ',
//  MAPNAME=SASCAGA
//SYSUT1   DD DISP=SHR,DSNAME=SASC.SAMPLE(SASCAMA)    INPUT MAP SOURCE //*
//*    EXECUTE THE SAS/C DSECT2C CATALOGED PROCEDURE
//*    USING THE PREVIOUSLY CREATED SYMBOLIC MAP TO CREATE
//*    A C STRUCTURE
//*
//  EXEC DSECT2C,PARM.D2C='SASCAGA -D'
//SYSLIB   DD DISP=SHR,DSN=your.symbolic.map.library
//SYSIN    DD *
 SASCAGA  DSECT
          COPY SASCAGA
          END
//D2C.D2COUT   DD DISP=SHR,DSN=your.c.header.library(SASCAGA)
```

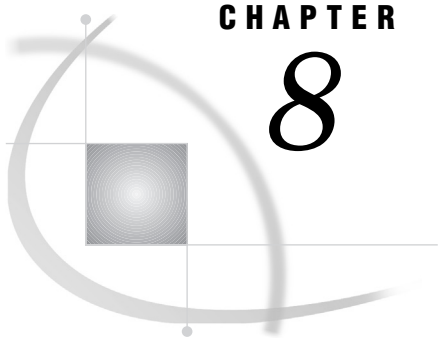
To use this map, the program would specify the following:

```
#include <sascaga.h>

main()
{
   EXEC CICS SEND MAP("SASCAGA") MAPONLY ERASE;
   EXEC CICS RETURN;
}
```

For additional information on this application, see the example in Appendix 1, "Examples," on page 99.

BMS Input/Output Operations

Use I/O commands (such as SEND MAP, RECEIVE MAP, SEND CONTROL, SEND TEXT, and SEND PAGE) to send the map, receive the map, handle text, and so on. These commands have no special SAS/C considerations. For details on using these commands in your program, see the IBM CICS Application Programmer's Reference appropriate for your site.



CHAPTER

8

Handling Files

Introduction **77**
Specifying Filenames and Access Methods **78**
General Filename Specification **78**
Access Method Parameters **78**
File Characteristics Amparms **78**
File Usage Amparms **79**
Working with Transient Data **80**
Positioning Transient Data Queues **80**
Specifying Transient Data Filenames **80**
Amparms for Transient Data **81**
recfm **81**
reclen **81**
blksize **82**
Default Values **82**
JES Spool File I/O **82**
JES Spool Filename Specification **83**
Specifying Amparms for JES Spool Files **83**
recfm **83**
reclen **83**
blksize **84**
Default Values **84**
Support for CICS File Control **84**
RID Field in ISSUE Commands **84**
DL/I Database Support **85**
SQL Database Support **89**

Introduction

This chapter provides more details on input/output operations and file handling for SAS/C CICS application programs. Topics covered include transient data control, JES (Job Entry Subsystem for OS/390) spool file support, and database support.

These items are of primary importance to the SAS/C programmer who is already familiar with CICS file-handling conventions. If you are not familiar with these conventions, refer to the relevant IBM documentation noted in the inside back cover of this book. It may also be useful to review Chapter 3, "I/O Functions," in the SAS/C Library Reference, Volume 1.

Specifying Filenames and Access Methods

SAS/C library I/O functions for sequential I/O are supported for CICS, except for UNIX style I/O functions. This set of SAS/C I/O functions is supported for transient data queues (both intrapartition and extrapartition) and JES spool files. Certain conventions must be observed for filenames and access methods when using transient data queues and JES spool files for SAS/C CICS applications.

General Filename Specification

The general form of a SAS/C filename is *style*:*name*, where the portion of the name before the colon defines the filename style, and the portion after the colon is the name of the file. For example, a style of **td** indicates the filename is a transient data queue name; the style **spl** indicates the filename is a JES spool file.

The *style*: part of the filename is optional. If no style is specified, the style is chosen by the compiler as follows:

- If you define the external variable **__style** with an initial value, then that value is used as the style. For example, if the initial value of **__style** is **td**, then the filename **logf** is interpreted as **td:logf**.
- If no initial value for **__style** is defined, the default is **td**.

For more information on the **__style** external variable, see Chapter 9, "Run-Time Argument Processing," in the SAS/C Compiler and Library User's Guide.

Access Method Parameters

When you open a file with **afopen**, **afreopen**, or **aopen**, you can optionally specify one or more access method parameters (amparms). These system-dependent options supply information about how the file will be processed or allocated. Amparms are character strings containing one or more specifications of the form amparm=value, separated by commas (for example, **"recfm=v,reclen=100"**). Amparms can be specified in any order and in uppercase or lowercase letters.

Amparms relevant to CICS describe file processing. This is a subset of the amparms supported by the library in other environments. Whenever reasonable, inapplicable amparms are ignored rather than rejected. The function descriptions for **afopen**, **afreopen**, and **aopen** in the SAS/C Library Reference, Volume 1 provide examples of typical usage.

File Characteristics Amparms

The file-processing amparms for CICS may be classified into two categories: file characteristics amparms and file usage amparms. The following amparms define file characteristics:

recfm=f|v|u
operating system record format

reclen=*nnn*|x
operating system record length

blksize=*nnn*
operating system block size

These amparms are used to specify a program's expectations for record format, maximum record length, and block size. If the file is not compatible with the program's specifications, the file is still opened, but a warning message is directed to the standard error file. **recfm** defines the file's expected record format.

- **recfm=f** indicates fixed-length records.
- **recfm=v** and **recfm=u** indicate varying-length records.

Note: The values of **recfm** must be specified exactly as shown (**f**, **v**, or **u**). No other characteristics are valid. △

reclen defines the maximum record length the program expects to read or write. The specification **reclen=x** (which is not permitted with **recfm** specifications other than **v**) indicates that there is no maximum record length. **blksize** specifies the maximum block size for the file, as defined by the operating system.

File Usage Amparms

The following amparms define file usage:

print=yes|no
file destined to be printed

page=*nnn*
maximum lines per page (with **print=yes**)

pad=no|null|blank
file padding permitted

These amparms allow a program to specify how a file will be used. A specification that cannot be honored may cause the open to fail, generate a warning message, or cause a failure later in execution, depending on the circumstances. **print=yes** or **print=no** is used to indicate whether the file is destined to be printed. If **print=yes** is specified, ANSI carriage-control characters are written to the first column of each record of the file to produce page formatting, if the file format permits this. In your C program, you can write the **'\f'** character to go to a new page and the **'\r'** character to perform overprinting. **print=yes** is allowed only for files accessed as text streams and whose open mode is **"w"**or **"a"**. If these conditions are satisfied, but the file characteristics do not support page formatting, a warning message is generated, and no page formatting occurs.

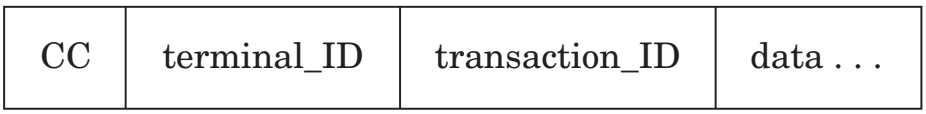
If **print=no** is specified, then the **'\f'** and **'\r'** characters in output data are treated as normal characters, even if the file characteristics will permit page formatting to take place.

page=*nnn* specifies the maximum number of lines that will be printed on a page. It is meaningful only for those files opened with **print=yes**, or for those for which **print=yes** is the default. It is ignored if it is specified for any other file. **pad** specifies how file padding is to be performed. **pad=blank** requests padding with blanks, **pad=null** requests padding with null characters, and **pad=no** requests no padding. If **pad=no** is specified, a record that requires padding is not written, and a diagnostic message is generated. **pad** is meaningful only for files with fixed-length records. For files accessed as text, pad characters are added as necessary to each output record, and removed from the end of each input record. For output files accessed as binary, padding only applies to the last record. For input files accessed as binary, padding is never performed.

Working with Transient Data

Transient data files can be either intrapartition or extrapartition data queues. Intrapartition queues are maintained by CICS in a VSAM data set and consist of varying-length records; output to intrapartition queues can also be used to automatically initiate transactions. The default record length for CICS intrapartition transient data queues is 480 bytes (the minimum possible VSAM control interval size of 512, minus 32 bytes of CICS control information).

Extrapartition queues are sequential data sets. They may have fixed- or varying-length records, and they may be blocked and contain ASA control characters. The format of output to extrapartition queues is the following:



CC is any ASA carriage control (if the file is defined as **`recfm=a`**).

For high-level programming languages running on CICS, the standard files (**`stderr`** and **`stdout`**) are usually directed to extrapartition transient data queues. Attempts to read from **`stdin`** will fail unless the default name is redirected.

Positioning Transient Data Queues

Although transient data queues cannot be positioned, certain input and output operations are still possible for both extrapartition and intrapartition queues. Extrapartition queues used for output can be positioned to EOF (end of file) because output to such files always adds new records to the end of the data set. Thus, these files are always positioned at EOF, which is the only valid position for extrapartition queues. Similarly, input extrapartition queues can be rewound, but only because they are always positioned at the start of the file.

Intrapartition queues can be used for both read and write operations. Records are logically read in the sequence they were written, and output always adds records to the end of the queue. This means that, in effect, intrapartition queues can be rewound because they are always positioned at the start of the file for input, and they can also be positioned to EOF because output always adds new records to the end of the file.

Note: Transient data reads are always destructive; that is, after a record has been read, it cannot be read again. △

Specifying Transient Data Filenames

A transient data filename is a four-character symbolic name associated with a specific destination, as defined in the CICS destination control table (DCT). The name can be in uppercase or lowercase, but it is translated to uppercase during processing. The name may refer to an indirect data destination but not to a remote destination. The default destinations associated with the standard files are

File	Destination
stdout	**td:saso**
stderr	**td:sase**
stgrpt	**td:sasr**

stgrpt is used for run-time storage analysis and usage reports when the **=storage** run-time argument is specified. Some examples of specifying transient data filenames follow:

- Print "hello world" on stdout:

```
printf("hello world");
```

- Open TD queue "logf":

```
f_ptr = fopen("td:logf", "a+");
```

- Open and read from TD queue "logf":

```
f_ptr = fopen("logf", "a+");
fread(string, 1, count, f_ptr);
```

- Redirect stdout to the TD queue "cssl":

```
char *_stdonm = "td:cssl";
```

Amparms for Transient Data

Extrapartition transient data files always exist with predefined file characteristics as they are preallocated and defined to CICS in the DCT. Specifying an amparm will not change the characteristics of an external file; rather, amparms are used to state a program's expectation of the file.

recfm

When using this amparm, consider these points:

- An open operation will fail if **recfm=f** is specified, but the file does not have fixed-length records.
- An open operation will fail if **recfm=v** or **recfm=u** is specified for an extrapartition file that is opened for output but is actually an F-format file.
- No diagnostic is generated if **recfm=v** or **recfm=u** is specified for a read-only file.
- **recfm=u** is ignored for intrapartition queues and is treated as **recfm=v** for extrapartition queues.
- Intrapartition queues are always treated as RECFM V files.

reclen

When using this amparm, consider these points:

- An open operation will fail for RECFM F files and for RECFM V output files if, for extrapartition queues, **reclen=*nnn*** is specified but the actual file's LRECL does not match this specification.
- An open operation will fail for RECFM V input files if the specified record length is greater than the actual LRECL of the file.

- **reclen=x** is not supported for transient data queues.
- A program's **reclen** specification is compared to the value of LRECL–4 for a V-format file, not to the LRECL itself. For a file with carriage-control characters, the **reclen** specification is compared to LRECL–1. Furthermore, extrapartition queues (which are prefixed with four-character terminal and transaction identifiers) are compared to LRECL–8. For example, an extrapartition VBA file with an LRECL of 133 can have at most a **reclen** specification of 120 (120 from the application + 8 for termid/tranid + 4 for V-format control information + 1 for carriage control).

blksize

When using this amparm, remember that if the actual block size of an extrapartition queue is greater than the size specified by **blksize=*nnn***, a warning will be generated; otherwise, this specification will be ignored.

Default Values

For extrapartition queues, default values for file characteristics amparms (**recfm**, **reclen**, and **blksize**) are derived from the values defined in the CICS DCT. For intrapartition queues, the following default values are assigned to the file characteristics amparms:

recfm	v
reclen	480
blksize	480

JES Spool File I/O

A SAS/C program can retrieve files from the JES spool, write a file directly to the JES spool, and send a file to a remote destination via systems connected to a JES/RSCS network. This feature makes it easy for you to generate and review output because you don't have to locate and then browse the sequential data set where extrapartition transient data output is usually sent.

Spool files consist of variable-length records up to 32K–8 bytes long. They do not have record format, record length, and block size characteristics. Spool files may have a JES class designation, and they may contain carriage-control characters. Spool file I/O is strictly sequential access. Like transient data files, essentially no positioning is available (output files are always positioned to EOF; input files are always positioned to the start of the file).

Input from JES is single-threaded; that is, only one transaction at a time in a CICS region may use the CICS input interface. However, many transactions can concurrently create output spool files. Input spool files are closed with a disposition of DELETE, unless an error occurs during processing, in which case the file is closed with a disposition of KEEP. If an input file is closed with the disposition of KEEP, the file remains on the spool and can be retrieved on a subsequent input operation.

Any open spool files are automatically closed by the EXEC CICS SYNCPOINT command. To avoid problems, issue the **fclose** function for any open spool file before issuing this command. Consult the IBM documentation on the CICS interface to JES for more details.

JES Spool Filename Specification

A spool filename is of the form *userid.node*<.*class*>, where *userid* and *node* are one to eight alphanumeric or national ($, #, or @) characters. An asterisk (*) may be specified for *userid* and *node* to direct an output file to the local spool. *class* is an optional alphanumeric character; the default is class A. For input spool files, *userid* is an external writer name that must begin with the same first four characters as the VTAM APPLID of the CICS system. See the IBM JCL reference manual for more information about specifying external writer names, userids, and node names. The following are examples of how to specify spool filenames:

- Redirect **stdout** to a userid at another node:

```
char *_stdonm = "spl:sascuser.vm";
printf("hello from CICS");
```

- Read a record from a spool file:

```
FILE *fptr;

fptr = fopen("spl:cicstest.*", "rb");
len  = afread(record, 1, 255, fptr);
```

- Submit a job for execution at the node named **OS/390**:

```
fptr = afopen("spl:intrdr.OS/390.a", "w",
       "seq", "recfm=f,reclen=80");

fprintf
(fptr, "%s\n", "//BR14 JOB (acct),pgmr_name");
fprintf
(fptr, "%s\n", "// EXEC PGM=IEFBR14");
fprintf(fptr, "%s\n", "/*EOF");
```

Specifying Amparms for JES Spool Files

The file characteristics amparms provide advice to the library about what file characteristics to expect. JES spool files may have either fixed- or varying-length records; however, files on the spool are actually formatted as variable-length records. The library will process a spool file according to the specifications of the file characteristics amparms.

Note: The defaults of **recfm=v**/**reclen=255** allow a program to read and write equal-length records just as easily as variable-length records. △

recfm

Consider the following points when using this amparm:

- If **recfm=f** is specified, the program expects records of equal length.
- If **recfm=v** is specified, the program expects records of variable length.
- **recfm=u** is treated as **recfm=v**.

reclen

Consider the following points when using this amparm:

- The value of this amparm tells the library the maximum input or output record length to expect; this value is used internally by the library as a buffer size. Record length cannot exceed the CICS maximum of 32,760.
- For ESA versions of CICS, **`reclen=x`** is supported, which means that records can be of any length up to the CICS maximum.
- You cannot specify both **`reclen=x`** and **`recfm=f`**.

blksize

Block size can be specified, but it is ignored for JES spool files.

Default Values

Default values for JES spool file characteristics amparms are

`recfm`	v
`reclen`	255
`blksize`	not applicable

Support for CICS File Control

Traditional CICS commands for random access (such as READ, WRITE, and DELETE) and commands for sequential access (such as STARTBR and READNEXT) are supported. The only area for special consideration is the record identification (RID) field for certain commands.

RID Field in ISSUE Commands

In the ISSUE REPLACE, ISSUE ADD, ISSUE ERASE, and ISSUE NOTE commands, if the record identification field is a relative record number, the RIDFLD option is expected to be an **`unsigned int`** type. Otherwise, the option is expected to be a **`pointer`** type (presumably a pointer to a key).

For the READ, WRITE, DELETE, STARTBR, READNEXT, READPREV, and RESETBR commands, the RIDFLD option is generally expected to be a **`pointer`** type, that is, a pointer to a key.

Each command has a set of options that, if used, specifies whether the argument will be either a relative byte address or a relative record number and, therefore, will be an **`unsigned int`** type. Here's the list of options for each command:

Command	Options
READ	RBA, RRN, GENERIC, GTEQ
READNEXT	RBA, RRN
READPREV	RBA, RRN
RESETBR	RBA, RRN
STARTBR	RBA, RRN
WRITE	RBA, RRN, MASSINSERT

Note: When the MASSINSERT option of the WRITE command is used, the RIDFLD option is expected to be of type **`void *`**, rather than of the **`unsigned int`** type. △

DL/I Database Support

SAS/C programs can use the DL/I call-level interface to process IMS/VS databases. No special interfaces are required, and DL/I applications written using SAS/C code are not restricted from using any C language features or library functions. The DL/I user interface block (UIB) is defined in the **`dliuib.h`** header file, which looks like this:

```
struct DLIUIB {       /* user interface block   */
   void *uibpcbal;    /* PCB address list       */
   struct{            /*                        */
      char uibfctr;   /* return codes           */
      char uibdltr;   /* additional information */
   } uibrcode;        /* DL/I return codes      */
   unsigned short _;  /* (reserved)             */
};
```

The EXEC DLI command-level interface to DL/I databases is also fully supported on CICS. EXEC DLI commands can be freely interspersed in your C program. The run-time library automatically performs a DL/I initialization call on your program's behalf the first time an EXEC DLI command is executed. This permits DL/I calls to occur in dynamically loaded C routines. A DL/I interface block (DIB) is also automatically allocated by the library, and can be addressed via the global external variable **`__dibptr`**. The header file containing the DIB definition is included in your program by the SAS/C command translator. The DIB is defined as follows:

```
struct DIB {
   char DIBVER[2];
   char DIBSTAT[2];
   char DIBSEGM[8];
   char DIBFIL01;
   char DIBFIL02;
   char DIBSEGLV[2];
   short DIBKFBL;
   char DIBFIL03[6];
} *__dibptr;
```

For special considerations on using the DL/I command-level interface in CICS, see the IBM CICS Programmer's Reference Manual. For additional information on using C with DL/I and the call-level interface, see the SAS/C Programmer's Report Developing IMS-DL/I Applications in C with the SAS/C Compiler. A complete example follows:

```
#include <stdio.h>
#include <ctype.h>
#include <packed.h>
#include <signal.h>
#include <options.h>

typedef struct {
   char ssn[11];
   char name[40];
   char address1[30];
   char address2[30];
```

```
   char city[28];
   char state[2];
   char country[20];
   char zipcode[10];
   char home_phone[12];
   char work_phone[12];
   char _1[30];
} Customer_Segment;

typedef struct {
   char number[12];
   char balance[5];
   char date[8];
   char stmt_bal[5];
   char _3[8];
} Account_Segment;

typedef struct {
   int amount;
   char _1[2];
   char date[4];
   char time[4];
   char description[66];
} Debit_Segment;

typedef struct {
   int amount;
   char _1[2];
   char date[4];
   char time[4];
   char description[66];
} Credit_Segment;

static void print_DIB();
static void dump(char *, int);
static void print_customer(const Customer_Segment *);
static void print_account(const Account_Segment *);

static char ioarea[sizeof(Customer_Segment)];

#define STATUS(code)
        (memcmp(_dibptr->DIBSTAT, code, 2) == 0)

main()
{

   /* Schedule a PSB with the name: ACCTRCOB. */
   EXEC DLI SCHEDULE PSB(ACCTRCOB);

   /* unqualified SSA...                      */
   printf("\n\n -Get All Customer Segments-\n");

   EXEC DLI GET UNIQUE USING PCB(1)
            SEGMENT("CUSTOMER ") INTO(ioarea);
```

```
#if defined(DUMPS)
      print_DIB();
   #endif

   while (STATUS("  "))
{
       print_customer((Customer_Segment *)ioarea);

        EXEC DLI GET NEXT USING PCB(1)
                SEGMENT("CUSTOMER ") INTO(ioarea);

     #if defined(DUMPS)
       print_DIB();
    #endif    }

   /*  qualified SSA...                        */
   printf("\n\n -Get Specific Customer Segment-\n");

      EXEC DLI GET UNIQUE USING PCB(1)
               SEGMENT("CUSTOMER ") WHERE(CUSTZIP="26001-0000")
               SEGMENT("CHCKACCT ") INTO(ioarea);

   #if defined(DUMPS)
      print_DIB();
      dump(ioarea, sizeof(Account_Segment));
   #endif

   print_account((Account_Segment *) ioarea);

   exit(0);
}

static void print_customer(const Customer_Segment *ioarea)
{
    puts("Customer Segment Dump:\n");
    printf("   SSN: %.11s\n", ioarea->ssn);
    printf("   Name: %.40s\n", ioarea->name);
    printf("   Address 1: %.30s\n", ioarea->address1);
    printf("   Address 2: %.30s\n", ioarea->address2);
    printf("   City: %.28s\n", ioarea->city);
    printf("   State: %.2s\n", ioarea->state);
    printf("   Country: %.20s\n", ioarea->country);
    printf("   Zip code: %.10s\n", ioarea->zipcode);
    printf("   Home phone: %.12s\n", ioarea->home_phone);
    printf("   Work phone: %.12s\n", ioarea->work_phone);
    printf("   Filler: %.30s\n\n", ioarea->_1);
 }


static void print_account(const Account_Segment *ioarea)
{
    double d;
    puts("Account Segment Dump:\n");
```

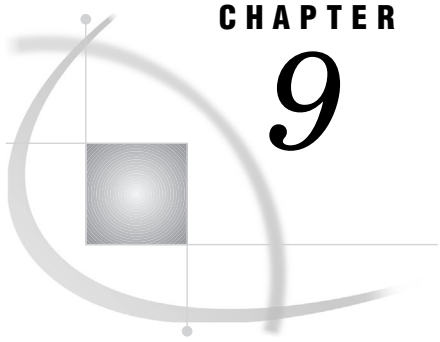
```
    printf("   Number: '%.12s'\n", ioarea->number);
    d = _pdval((char (*)[])ioarea->balance, 5, 2);
    printf("   Balance: %g\n", d);
    printf("   Date: '%.8s'\n", ioarea->date);
    d = _pdval((char (*)[])ioarea->stmt_bal, 5, 2);
    printf("   Statement Balance: %g\n\n", d);
 }

static void print_DIB()
{
    puts("DIB Dump:\n");
    printf("   Version : '%.2s'\n", _dibptr->DIBVER);
    printf("   Status  : '%.2s'\n", _dibptr->DIBSTAT);
    printf("   Status1 : '%.2x'\n", _dibptr->DIBSTAT[0]);
    printf("   Status2 : '%.2x'\n", _dibptr->DIBSTAT[1]);
    printf("   Seg name: '%.8s'\n", _dibptr->DIBSEGM);
    printf("   Fil  #1 : '%.c'\n", _dibptr->DIBFIL01);
    printf("   Fil  #2 : '%.c'\n", _dibptr->DIBFIL02);
    printf("   Seg ver : '%.2s'\n", _dibptr->DIBSEGLV);
    printf("   Key leng: '%.d'\n", _dibptr->DIBKFBL);
    printf("   Fil  #3 : '%.c'\n", _dibptr->DIBFIL03);
 }

static void dump(char *area, int length)
{
    int c, l;
    #define BYTES_PER_LINE 16

    printf("Data Dump - Length = %d\n\n", length);
    if (length % BYTES_PER_LINE)
       length = ((length + BYTES_PER_LINE-1) / BYTES_PER_LINE)
                                                    * BYTES_PER_LINE;
    for (l = 0; l < length; l += BYTES_PER_LINE)0
        printf("   +%0.4X  *", l);
        for (c = 0; c < BYTES_PER_LINE; c++)
           printf("%0.2X", area<l+c>);
        fputs("*  *", stdout);
        for (c = 0; c < BYTES_PER_LINE; c++)
           putchar(isprint(area[l+c]) ? area[l+c] : '.');
        puts("*");
     }    putchar('\n');
 }
```

Here is the JCL for this example:

```
//CICS    EXEC LCCCPCL,
//  PARM.CCP='DLI ',
//  PARM.C='NOEXC SNAME(DLISAMP) RENT '
//  PARM.LKED='LIST MAP RENT REUS XREF '
//CCP.SYSIN    DD DSN=userid.SASC.CCP(DLI$SAMP),DISP=SHR
//LKED.SYSLMOD DD DSN=system.CICSAPPL.LOADLIB,DISP=SHR
//*
```

SQL Database Support

The SAS/C CICS translator supports development of C language application programs containing embedded Structured Query Language (SQL) statements. For information describing linkage conventions used by SQL functions, useful C macros that should be used in SQL programs, and detailed instructions on running a sample SAS/C program, see the Programmer's Report Developing SQL Applications in C with the SAS/C Compiler.



CHAPTER

9

TCP/IP Socket Library Support for the CICS and Environment

Overview of TCP/IP **91**
Overview of the BSD UNIX Socket Library **91**
SAS/C Socket Library for TCP/IP **92**
TCP/IP Socket Library Support for CICS **92**
Resident Functions Supported under CICS **92**
Communications Functions **93**
takesocket Function **94**
takesocket Example **94**
I/O Functions **96**
Unsupported Configuration Information Functions **96**

Overview of TCP/IP

The Transmission Control Protocol/Internet Protocol (TCP/IP) provides a means of connecting existing computer networks of computers that run different operating systems and that are manufactured by different vendors. TCP/IP is designed to accommodate a large number of host computers and local networks.

The open, nonproprietary nature of TCP/IP and its global scope have made it popular among users of the UNIX environment. Standards for writing communications programs in C have also become widespread. The two most common standards are the BSD UNIX socket library interface and the UNIX System V Transport Layer Interface (TLI). The SAS/C Library currently implements the BSD UNIX socket library interface because it is somewhat more common than TLI and it has better support from the underlying communications software on OS/390 and CMS systems.

Overview of the BSD UNIX Socket Library

BSD UNIX communications programming is based on the original UNIX framework, with some additions and elaborations to account for the greater complexity of interprocess communications. In traditional UNIX file I/O, an application issues an **`open`** call, which returns a file descriptor (a small integer) bound to a particular file or device. The application then issues a **`read`** or **`write`** call that causes the transfer of stream data. At the end of communications, the application issues a **`close`** call to terminate the interaction. Because interprocess communication often occurs over a network, the BSD UNIX socket library accounts for the numerous variables of network I/O, such as network protocols, and for the semantics of the UNIX file system.

A *socket* is an end point for interprocess communication, in this case, over a network running TCP/IP. Using semantics that depend on the type of socket, sockets can

simultaneously transmit and receive data from another process. The socket interface can support a number of underlying transport mechanisms. Ideally, a program written with socket calls can be used with different network architectures and different local interprocess communication facilities with few or no changes. The SAS/C Compiler supports TCP/IP and the AF_INET Internet addressing family.

SAS/C Socket Library for TCP/IP

The SAS/C Socket Library is integrated with SAS/C support for UNIX file I/O to provide the same type of integration between file and network I/O that is available on BSD UNIX systems.

The SAS/C Socket Library for TCP/IP supports socket function calls for the CICS environment except for:

- socket functions used to access network configuration files, such as the **`/etc/hosts`** or **`/etc/service`** format files
- socket functions that use the resolver.

Socket function calls do not require any special link-editing considerations.

The SAS/C Socket Library relies on an underlying layer of TCP/IP communications software, such as IBM TCP/IP Version 2 for VM and OS/390. TCP/IP communications software handles the actual communications. The SAS/C Library adds a higher level of UNIX compatibility and provides integration with the SAS/C run-time environment.

The open architecture of the SAS/C Socket Library permits the use of TCP/IP products from different vendors. If a vendor provides the appropriate SAS/C transient library module, existing socket programs can communicate using the TCP/IP implementation specified during configuration of the system. A program compiled and linked with the SAS/C Socket Library at one site can be distributed to sites running different TCP/IP implementations. Also, any site can change TCP/IP vendors without recompiling or relinking its SAS/C applications. Refer to SAS/C Compiler and Library User's Guide for details on the SAS/C Socket Library.

TCP/IP Socket Library Support for CICS

The SAS/C Socket Library for TCP/IP supports all socket function calls for CICS. Socket function calls do not require any special link-editing considerations. Refer to the SAS/C Compiler and Library User's Guide for detailed information on compiling and linking CICS applications.

To use these socket function calls, you must have the CICS to TCP/IP Socket Interface feature available with TCP/IP Version 2, Release 2 for OS/390 from International Business Machines Corporation.

Resident Functions Supported under CICS

The following TCP/IP resident functions are supported under CICS without restrictions:

_getlong

_getshort

htoncs

htonl

htons

inet_addr

inet_lnaof

inet_makeaddr

inet_network

inet_netof

inet_ntoa

ntohcs

ntohl

ntohs

putlong

putshort

setsockimp

Communications Functions

The following TCP/IP communications functions are supported under CICS:

accept

bind

close

connect

fcntl

getclientid

gethostbyaddr

gethostbyname

gethostid

gethostname

getpeername

getsockname

getsockopt

givesocket

ioctl

listen

select

setsockopt

shutdown

socket

takesocket

takesocket Function

The **takesocket** function takes a socket descriptor from a donor process. The socket descriptor must be a SAS/C socket descriptor. If the descriptor is obtained directly from a donor process that is not part of the SAS/C socket library, the function may require TCP/IP vendor-dependent transformation.

Specifically for CICS, when the CICS TCP/IP listener task (CSKL) passes a socket descriptor as member **give_take_socket** of the TCPSOCKET_PARM data, use a **#define** statement to define the symbol **__IBM_TCPIP** within the program or in the compilation parameters. This definition makes available the macro FD_FROM_IBM_TCPIP(s) to convert an IBM socket descriptor to the equivalent SAS/C descriptor for **takesocket** to use. The inverse macro FD_TO_IBM_TCPIP(s) is also available.

The following structure maps out the TCPSOCKET_PARM that the CICS listener (supplied by IBM) passes as FROM data in the EXEC CICS START command when it initiates a SAS/C TCP/IP CICS transaction application program. These data may be obtained by coding an EXEC CICS RETRIEVE command in the program.

```
struct TCPSOCKET_PARM{
   int give_take_socket;    /* socket number given by listener */
   char lstn_name[8];       /* listener name                   */
   char lstn_subname[8];    /* listener subname                */
   char client_in_data[36]; /* client passed data              */
   struct sockaddr_in
        sockaddr_in_parm;   /* Internet socket address         */
};
```

If **takesocket** succeeds, it returns a nonnegative SAS/C socket descriptor. If it fails, it returns a –1 and sets **errno** to indicate the type of error. For more information, see SAS/C Library Reference, Volume 2 and the IBM publication TCP/IP Sockets Interface for CICS.

takesocket Example

This program acts as a Server to take a socket from the CICS listener transaction that is supplied by IBM; then it writes date/time information to the socket. The SAS/C CICS translator is required to translate this source before compilation.

```
#define __IBM_TCPIP 1

#include <cics.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

main()
{
   int cs;                    /* client socket returned by takesocket */
   int s;                     /* socket passed by the listener */
```

```
struct clientid clientid;  /* return struct for clientid */
                           /* information */
char *cl_info;

                           /* variables used to obtain the time */
char *p;
int len;
time_t t;


int n_times;               /* loop count */



char outbuf[128];          /* buffer for outgoing time string */

   /* Map of parms passed to program by CICS listener via       */
   /*  "EXEC CICS START" and picked up by "EXEC CICS RETRIEVE". */
struct TCPSOCKET_PARM{
   int give_take_socket;
   char lstn_name[8];
   char lstn_subname[8];
   char client_in_data[36];
   struct sockaddr_in sockaddr_in_parm;
} csp;                     /* cics_socket_parm */

short csp_len = (short) sizeof(csp);

EXEC CICS RETRIEVE INTO(&csp) LENGTH(csp_len); /* Get parms. */

   /* Build required CICS format clientid for takesocket. */
memset(clientid, 0, sizeof(clientid));    /* zero out clientid */
clientid.domain=AF_INET;
memcpy(clientid.name, csp.lstn_name, 8);
memcpy(clientid.subtaskname, csp.lstn_subname, 8);

printf("Passed clientid info:  "
       "Domain=%d, Name=<%.8s>, Task=<%.8s>, Resv=<%.20s>\n",
       clientid.domain, clientid.name, clientid.subtaskname);

   /* Convert IBM socket number to SAS/C socket number base. */
s = FD_FROM_IBM_TCPIP(csp.give_take_socket);

   /* Take the passed client socket; takesocket returns a local */
   /*  socket number enabling us to write to the client.        */
cs = takesocket(&clientid, s);

   /* Check takesocket rc. */
if (cs == -1){
   perror("takesock() call failed");
   exit(EXIT_FAILURE);
}
```

```
   n_times = 2;
   while (n_times--){
         /* Send the time to the client. Clients */
         /* expect the string to be in ASCII.    */
      time(&t);                  /* machine-readable time */
      p = ctime(&t);             /* human-readable time */
         /* Convert to ASCII if necessary. */
      for (len=0; p[len] && len<sizeof(outbuf); len++)
         outbuf[len] = htoncs(p[len];
      outbuf[len+1] = htoncs('\n');  /* Send a new line. */

      if (write(cs, outbuf, len)==-1){
         perror("write() failed");
         printf("Client IP address: %s\n",
                 inet_ntoa(csp.sockaddr_in_parm.sin_addr));
         return EXIT_FAILURE;
      }
    }
   close(cs);
   return EXIT_SUCCESS;          /* Avoid compilation warnings. */
}
```

I/O Functions

For the following I/O functions, IBM CICS TCP/IP imposes a 32768-byte limit for data transmission:

read

readv

recv

recvfrom

recvmsg

send

sendmsg

sendto

write

writev

The library does not check to see if the 32768-byte limit is exceeded, but the underlying CICS API socket rejects the request if the limit is exceeded. The IBM interface requires the vector I/O functions **`readv`**, **`writev`**, **`sendmsg`**, and **`recvmsg`** to allocate (using **`malloc`**) a buffer equal in size to the combined sum of the vector lengths.

Unsupported Configuration Information Functions

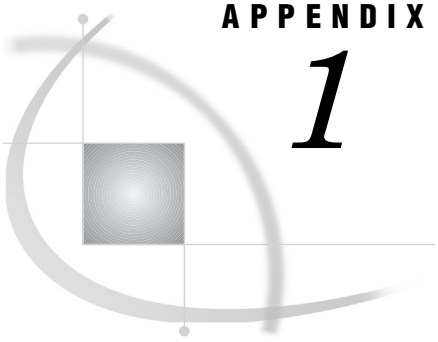
The following TCP/IP configuration information functions are presently not supported under CICS. Dummy placeholder functions exist for them that return either

NULL, –1, or no return value, in keeping with the function's return value declaration of pointer, **int**, or **void**.

Table 9.1 Unsupported Functions under CICS

Function	Return Value
gethostent	NULL
sethostent	–1
endhostent	void
herror	void
getservbyname	NULL
getservbyport	NULL
getservent	NULL
setservent	void
endservent	void
getprotobyname	NULL
getprotobynumber	NULL
getprotoent	NULL
setprotoent	void
endprotoent	void
getnetbyname	NULL
getnetbyaddr	NULL
getnetent	NULL
setnetent	void
endnetent	void
res_mkquery	–1
res_init	–1
res_send	–1
dn_comp	–1
dn_expand	–1

herrno and **_res** are not supported under CICS. The use of these functions will probably cause an ASRA error. **socketpair** is also unsupported.



APPENDIX

1

Examples

Introduction **99**
SASCMNU: Display the Main Menu **100**
SASCALL: Perform Inquiry and Update Functions **100**
SASCBRW: Perform Browse Function **109**

Introduction

This appendix discusses and provides source listings for some of the example programs available in the SASC.SAMPLE data sets under OS/390 and the LCSAMPLE MACLIB under CMS.

The sample programs comprise an application that shows a main selection menu on the display enabling the user to inquire, add, update, and browse records.

The example application is provided in source code only. Before executing the sample programs, you need to construct and install executable load modules. Also, you need to define the maps, programs, and transactions to the CICS environment.

Basic Mapping Support (BMS) maps are an integral part of this application. The source code for these maps is also supplied in the SASC.SAMPLE library under OS/390 and the LCSAMPLE MACLIB under CMS. These maps are called SASCAMA, SASCAMB, and SASCAMC. Chapter 7, "Terminal Control and Basic Mapping Support," provides additional information about creating BMS maps for SAS/C applications.

The examples assume that the traditional CICS FILEA data set has been installed at your site. Please consult your CICS systems manager or the CICS Installation and Operations manual if the FILEA example is not installed.

Perform the following tasks to construct the sample application. After these tasks are complete, you can execute the sample application by keying the transaction identifier SMNU in the top-left corner of the CICS terminal screen and pressing the EN TER key. From there, follow the instructions presented on the menu to direct you through the rest of the example.

1. Assemble the BMS maps and link-edit them into your CICS application map library. You also need to create C language structures that define those maps. The C structures can be stored in your C application header files, or they can be placed in the standard header file library. To accomplish this, you can use the IBM cataloged procedure DFHMAPS and the SAS Institute cataloged procedure DSECT2C. See Chapter 7, "Terminal Control and Basic Mapping Support," on page 71 for more information on these steps.
2. Translate, compile, and link-edit the example programs. The resulting load modules need to be available to CICS on the DFHRPL DD concatenation. The SAS Institute cataloged procedure LCCCPCL, CLIST's LCCCP and CLK370, or CMS EXEC's LCCCP and CLINK can be used to accomplish this task. See Chapter 5,

"Preprocessing, Compiling, and Linking," on page 35 for more information on translating, compiling, and linking.

3 Define the programs, maps, and transactions to CICS. Sample CICS table-entry macros for these resources are provided in the SASC.SAMPLE library as member name SASCTBLS. These macros can be copied into your own tables, then reassembled and reinstalled; or the macros can serve as information when using Resource Definition Online (RDO, transaction identifier CEDA) to define and install the resources.

As you look over these examples, note the way exceptional conditions are handled. In C, functions are executed when a HANDLE CONDITION request is raised rather than using branches that depend on response code values.

SASCMNU: Display the Main Menu

The display program, SASCMNU, corresponds to the IBM DFH$*x*MNU programs (where *x* is A for Assembler, P for PL/I, and C for COBOL). It displays the main selection menu for the FILEA application example on the terminal. From this menu, all other functions can be selected.

```
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                                                                      |
|                   S A S / C  S A M P L E                             |
|                                                                      |
|          NAME: SASCMNU                                               |
|       PURPOSE: Traditional CICS FILEA sample program                 |
| INSTALLATION: Assemble the associated BMS map (SASCAMA)              |
|               and then use DSECT2C to create a C header that         |
|               defines the map. Use the sample PPT input              |
|               (SASCTBLS) to define the resource CICS.                |
|       COMPILE: Use the CICS translator procedure LCCCPCL              |
|       EXECUTE: Via transaction SMNU                                  |
|        USUAGE:                                                       |
| SYSTEM NOTES:                                                        |
|                                                                      |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
#include <sascaga.h>
void main()
main()
{
   EXEC CICS SEND MAP("SASCAGA") MAPONLY ERASE;
   EXEC CICS RETURN;
}
```

SASCALL: Perform Inquiry and Update Functions

The SASCALL program corresponds to the IBM programs DFH$*x*ALL (where *x* has the same values as discussed in the previous section). It is the program that does the individual record inquiry, adds new records, and updates old ones.

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                                                                      |
```

```
|                    S A S / C  S A M P L E                |
|                                                          |
|         NAME: SASCALL                                    |
|      PURPOSE: Traditional CICS FILEA sample program      |
| INSTALLATION: Assemble the associated BMS maps           |
|               (SASCAMA, SASCAMB) and then use DSECT2C to |
|               create C headers that define those maps.   |
|               Use the sample PPT input (SASCTBLS) to     |
|               define the resources to CICS.              |
|      COMPILE: Use the CICS translator procedure LCCCPCL  |
|      EXECUTE: Via transaction SMNU                       |
|        USAGE:                                            |
| SYSTEM NOTES:                                            |
|                                                          |
+ - - - - - - - - - - -- - - - - - - - - - - - - - -- -- - - */
#include <dfhbmsca.h>   /* useful BMS definitions */

   /* C struct defining Map A, produced by DSECT2C */
#include <sascaga.h>

   /* C struct defining Map B, produced by DSECT2C */
#include <sascagb.h>
#include <string.h>     /* standard string header file */

   /* general-purpose variables */
char messages[39];      /* message-holding area */

   /* The following two strings are used to validate input data. */
char *valid_name_chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ .-'";
char *valid_amount_chars = "0123456789.$";

short comlen;           /* length of the CICS COMMAREA */
char keynum[6];         /* record key, the account number */
int i;

   /* FILEA record layout */
struct FILEA{
   char stat;
   char numb[6];
   char name[20];
   char addrx[20];
   char phone[8];
   char datex[8];
   char amount[8];
   char comment[9];
} filea;


struct FILEA commarea;  /* definition of the COMMAREA */

   /* function prototypes */
void data_error(void);
void map_build(void);
void map_send(void);
```

```
void cics_control(void);
void notmodf(void);
void duprec(void);
void badleng(void);
void badchars(void);
void notfound(void);
void mfail(void);
void errors(void);
void smnu(void);

   /* The main function begins here. */
main(struct EIBLK *eib_pointer, void *commarea_pointer)

   /* Note that __commptr, __eibptr, and __dibptr are             */
   /* implementation-provided global external variables available */
   /* to all SAS/C CICS programs. You can also, as demonstrated   */
   /* in this program, define your own pointers to these commonly */
   /* used areas.                                                 */
{
      /* Second time through? */
   if (commarea_pointer) goto read_input;

   /* If first time through, present the initial screen. */
   EXEC CICS HANDLE CONDITION ERROR(errors) MAPFAIL(mfail);

   EXEC CICS RECEIVE MAP("SASCAGA");

      /* A key must be provided. */
   if (memcmp(sascaga.keyl,"\0 \0",2) == 0){
      badleng();         /* no return from badleng() */
   }

      /* Verify that all six digits of the key are numeric. */
   for(i=0;i<=5;i++){
      if (isdigit((int) sascaga.KEYI[i]) == 0){
         strcpy(messages,"ACCOUNT NUMBER MUST BE NUMERIC");
         smnu();          /* There is no return from smnu(). */
      }
   }

      /* Save the key. */
   memcpy(keynum,sascaga.KEYI,sizeof(sascaga.KEYI));
   memset(&sascagb,'\0',SASCAGBE);  /* Clear the map. */

      /* Is this an add? */
   if (memcmp(eib_pointer->EIBTRNID,"SADD",4) == 0){

         /* Set up Map B for an add operation. */
      strcpy(sascagb.titleo,"FILE ADD");
      strcpy(sascagb.msg3o,"ENTER DATA AND PRESS ENTER KEY");
      memcpy(&sascagb.numbo,sascaga.KEYI,sizeof(sascaga.KEYI));
      memcpy(&commarea.numb,sascaga.KEYI,sizeof(sascaga.KEYI));
      sascagb.amounta = DFHBMUNN;
```

```
      /* The COMMAREA contains only status and account no. */
   comlen = 7;
   map_send();         /* Display Map B. */
   cics_control();     /* Pass control back to CICS. */
}
else

      /* Is this a query? */
   if (memcmp(eib_pointer->EIBTRNID,"SINQ",4) == 0){

         /* Try to read the record specified, but be */
         /* prepared if it is not found.             */
      EXEC CICS HANDLE CONDITION NOTFND(notfound);

      EXEC CICS READ DATASET("FILEA") INTO(&filea)
                RIDFLD(keynum) LENGTH(sizeof(struct FILEA))
                KEYLENGTH(sizeof(keynum));

         /* The record was found, so set up Map B for a query. */
      strcpy(sascagb.titleo,"FILE INQUIRY");
      strcpy(sascagb.msg3o,"PRESS ENTER TO CONTINUE");

         /* Move the data from the record */
         /* area into the map fields.     */

      map_build();

         /* Protect all the fields. */
      sascagb.namea    = DFHBMPRO;
      sascagb.addra    = DFHBMPRO;
      sascagb.phonea   = DFHBMPRO;
      sascagb.datea    = DFHBMPRO;
      sascagb.amounta  = DFHBMPRO;
      sascagb.commenta = DFHBMPRO;
      map_send();       /* Display Map B. */

         /* Return to CICS and display the main menu again. */
      EXEC CICS RETURN TRANSID("SMNU");
}
else

         /* Is this an update? */
      if (memcmp(eib_pointer->EIBTRNID,"SUPD",4) == 0){

            /* Try to read the record specified, but be */
            /* prepared if it is not found.             */
         EXEC CICS HANDLE CONDITION NOTFND(notfound);

         EXEC CICS READ DATASET("FILEA") INTO(&filea)
                   RIDFLD(keynum) LENGTH((short) sizeof(struct FILEA))
                   KEYLENGTH((short) sizeof(keynum));

         /* The record was found, so set up Map B for an update. */
      strcpy(sascagb.titleo,"FILE UPDATE");
```

```
            strcpy(sascagb.msg3o,"CHANGE FIELDS AND PRESS ENTER");

               /* Save the record in the COMMAREA for later comparison. */
            memcpy(&commarea,&filea,sizeof(struct FILEA));

            map_build();      /* Move the data from the record */
                              /*  area into the map fields.    */

            map_send();       /* Display Map B. */

            comlen = 80;      /* The COMMAREA contains the entire record. */

            cics_control();   /* Pass control back to CICS. */
         }
         else{
            errors();         /* The transaction ID was not recognized. */
                              /* There is no return from errors().      */
         }

      /* Things begin to happen the second time through. */
read_input:

      /* Retrieve the COMMAREA. */
memcpy(&commarea,commarea_pointer,sizeof(commarea));

      /* Prepare to handle error conditions. */
EXEC CICS HANDLE CONDITION MAPFAIL(notmodf) DUPREC(duprec)
                           ERROR(errors) NOTFND(notfound);

      /* Read the input map. */
EXEC CICS RECEIVE MAP("SASCAGB");

      /* Is this an update? */
if (memcmp(eib_pointer->EIBTRNID,"SUPD",4) == 0){

         /* Try to read the specified record with intent to update. */
      EXEC CICS READ UPDATE DATASET("FILEA") INTO(&filea)
                RIDFLD(commarea.numb)
                LENGTH((short) sizeof(struct FILEA))
                KEYLENGTH((short) sizeof(commarea.numb));

         /* If the record just read does not exactly match the */
         /* one read during the first pass, another user must  */
         /* have updated the record while this transaction was */
         /* rolled out.  If so, refuse to update the record.   */
      if (memcmp(&filea,&commarea,sizeof(struct FILEA))){

            /* Refuse to update and prepare screen for another attempt. */
         strcpy(sascagb.msg1o,"RECORD UPDATED BY OTHER USER, TRY AGAIN");
         sascagb.msg1a = DFHBMASB;
         sascagb.msg3a = DFHPROTN;

            /* Move the data from the record area into the map fields. */
         map_build();
```

```
         /* Resend only the data portion of the map. */
      EXEC CICS SEND MAP("SASCAGB") DATAONLY;

         /* Make sure that the same record is used */
         /* during the next pass.                  */
      commarea = filea;
      comlen = 80;
      cics_control();    /* Pass control back to CICS. */
   }

   else{

         /* It is OK to update the record.  Note on the next screen. */
      filea.stat = 'U';
      strcpy(messages,"RECORD UPDATED");
   }
}
      /* Is this an add? */
   else
      if (memcmp(eib_pointer->EIBTRNID,"SADD",4) == 0) {

            /* Yes, it's an add.  Note on the next screen. */
         filea.stat = 'A';
         strcpy(messages,"RECORD ADDED");
      }
   else{

         /* The transaction ID was not recognized. There is */
         /* no return from errors().                        */
      errors();
   }

   /* Check to see if any of the fields were modified. */
if (memcmp(sascagb.namel,"\0 \0",2)    == 0 &&
    memcmp(sascagb.addrl,"\0 \0",2)   == 0 &&
    memcmp(sascagb.phonel,"\0 \0",2)  == 0 &&
    memcmp(sascagb.datel,"\0 \0",2)   == 0 &&
    memcmp(sascagb.amountl,"\0 \0",2) == 0 &&
    memcmp(sascagb.commentl,"\0 \0",2) == 0){

       /* No fields were modified, so take no action. */
    notmodf();    /* There is no return from notmodf(). */
}

   /* Is this an add? */
if (memcmp(eib_pointer->EIBTRNID,"SADD",4) == 0){

      /* It's an add, so validate the name field. */
   for(i=0;i < *(short *) sascagb.namel ;i++){
      if (strchr(valid_name_chars,sascagb.NAMEI[i]) == 0){
         data_error();  /* There is no return from data_error. */
      }
   }
```

```
      /* Enforce the account number to be what */
      /* was originally stated.                */
   memcpy(&filea.numb,&commarea.numb,sizeof(commarea.numb));
   }
else

      /* Is this an update? */
   if (memcmp(eib_pointer->EIBTRNID,"SUPD",4) == 0){

         /* It's an update, so validate the name */
         /* field if it was changed.             */
      if (memcmp(sascagb.namel,"\0 \0",2) != 0){
         for(i=0;i < *(short *) sascagb.namel ;i++){
            if (strchr(valid_name_chars,sascagb.NAMEI[i]) == 0){
               data_error();  /* There is no return from data_error. */
            }
         }
      }

         /* Also, validate the amount field if it was changed. */
      if (memcmp(sascagb.amountl,"0",2) != 0){
         for(i=0;i < *(short *) sascagb.amountl ;i++){
            if (strchr(valid_amount_chars,sascagb.AMOUNTI[ ]= 0){
               data_error();  /* There is no return from data_error. */
            }
      }
   }
}

   /* Update the record area with any fields that */
   /* were changed on screen.                     */
if (memcmp(sascagb.namel,"\0 \0",2) != 0){
   memcpy(&filea.name,sascagb.nameo,sizeof(sascagb.nameo));
}
if (memcmp(sascagb.addrl,"\0 \0",2) !=0){
   memcpy(&filea.addrx,sascagb.addro,sizeof(sascagb.addro));
}
if (memcmp(sascagb.phonel,"\0 \0",2) !=0){
   memcpy(&filea.phone,sascagb.phoneo,sizeof(sascagb.phoneo));
}
if (memcmp(sascagb.datel,"\0 \0",2) !=0){
   memcpy(&filea.datex,sascagb.dateo,sizeof(sascagb.dateo));
}
if (memcmp(sascagb.amountl,"\0 \0",2) !=0){
   memcpy(&filea.amount,sascagb.amounto,sizeof(sascagb.amounto));
}
else{

      /* For an add, fill in the default amount. */
   if (memcmp(eib_pointer->EIBTRNID,"SADD",4) == 0){
      strcpy(filea.amount,"$0000.00");
   }
}
if (memcmp(sascagb.commentl,"\0 \0",2) !=0){
```

```
   memcpy(&filea.comment,sascagb.commento,sizeof(sascagb.commento));
}

   /* If this is an update, rewrite the record to the file. */
   /* If this is an add, write the record to the file for   */
   /* the first time.                                       */
if (memcmp(eib_pointer->EIBTRNID,"SUPD",4) == 0){
   EXEC CICS REWRITE DATASET("FILEA") FROM(&filea)
                     LENGTH((short) sizeof(filea));
}
else{
   EXEC CICS WRITE DATASET("FILEA") FROM(&filea)
                   RIDFLD(commarea.numb) LENGTH((short) sizeof(filea));
}
smnu();            /* There is no return from smnu(). */
}                  /* End of main() */

   /* The data_error function is called when invalid */
   /* data is entered in one of the Map B fields.    */
   /* The data error message is added, and the       */
   /* screen is refreshed for another attempt.       */
void data_error()
{
   sascagb.msg3a    = DFHBMASB;
   strcpy(sascagb.msg3o,"DATA ERROR - CORRECT AND PRESS ENTER");
   sascagb.amounta  = DFHUNNUM;
   sascagb.namea    = DFHBMFSE;
   sascagb.addra    = DFHBMFSE;
   sascagb.phonea   = DFHBMFSE;
   sascagb.datea    = DFHBMFSE;
   sascagb.commenta = DFHBMFSE;

   EXEC CICS SEND MAP("SASCAGB") DATAONLY;

   /* Note that in the following statement, the */
   /* implementation-defined global external    */
   /* variable _eibptr is used instead of the   */
   /* application-defined local variable        */
   /* eib_pointer from main().                  */
if (memcmp(_eibptr->EIBTRNID,"SADD",4) == 0){
   comlen = 7; /* If it's an add, only the account number is passed. */
}
else{
   comlen = 80; /* Otherwise, the entire record is passed. */
}
cics_control();      /* Pass control back to CICS. */
}                    /* end of data_error() */

   /* This function moves all the fields from the */
   /* record area into the map areas.             */
void map_build(void)
{
   memcpy(sascagb.numbo,filea.numb,sizeof(filea.numb));
   memcpy(sascagb.nameo,filea.name,sizeof(filea.name));
```

```
   memcpy(sascagb.addro,filea.addrx,sizeof(filea.addrx));
   memcpy(sascagb.phoneo,filea.phone,sizeof(filea.phone));
   memcpy(sascagb.dateo,filea.datex,sizeof(filea.datex));
   memcpy(sascagb.amounto,filea.amount,sizeof(filea.amount));
   memcpy(sascagb.commento,filea.comment,sizeof(filea.comment));
   return;
}      /* end of map_build() */



   /* This function displays Map B on the screen. */
void map_send(void)
{
   EXEC CICS SEND MAP("SASCAGB") ERASE;
return;
}      /* end of map_send() */

   /* This function returns control to CICS.  It causes */
   /* the same transaction to be reinitiated with a     */
   /* COMMAREA.  This is commonly known as a            */
   /* pseudo-conversational rollout.                    */
void cics_control(void)
{

   /* Note that in the following statement, the */
   /* implementation-defined global external    */
   /* variable _eibptr is used instead of the   */
   /* application-defined local variable        */
   /* eib_pointer from main()/                  */
EXEC CICS RETURN TRANSID(_eibptr->EIBTRNID) COMMAREA(&commarea)
                LENGTH(comlen);
}  /* end of cics_control() */

   /* This function prepares an error message and calls smnu(). */
void notmodf(void)
{
   strcpy(messages,"RECORD NOT MODIFIED");
   smnu();     /* There is no return from smnu(). */
}              /* end of notmodf() */

   /* This function prepares an error message and calls smnu(). */
void duprec(void)
{
   strcpy(messages,"DUPLICATE RECORD");
   smnu();     /* There is no return from smnu(). */
}              /* end of duprec() */

   /* This function prepares an error message and calls smnu(). */
void badleng(void)
{
   strcpy(messages,"PLEASE ENTER AN ACCOUNT NUMBER");
   smnu();     /* There is no return from smnu(). */
}              /* end of badleng() */
```

```
   /* This function prepares an error message and calls smnu(). */
void badchars(void)
{
   strcpy(messages,"ACCOUNT NUMBER MUST BE NUMERIC");
   smnu();     /* There is no return from smnu(). */
}              /* end of badchars() */

   /* This function prepares an error message and calls smnu(). */
void notfound(void)
{
   strcpy(messages,"INVALID NUMBER - PLEASE REENTER");
   smnu();     /* There is no return from smnu(). */
}              /* end of notfound() */

   /* This function prepares an informational message */
   /* and calls smnu().                               */
void mfail(void)
{
   strcpy(messages,"PRESS CLEAR TO EXIT");
   smnu();     /* There is no return from smnu(). */
}              /* end of mfail() */

   /* This function prepares an error message and calls smnu(). */
void errors(void)
{
   EXEC CICS DUMP DUMPCODE("ERRS"); /* Cause a transaction dump. */
   strcpy(messages,"TRANSACTION TERMINATED");
   smnu();     /* There is no return from smnu(). */
}              /* end of errors() */

   /* This function displays the main menu on the screen. */
void smnu(void)
{
   memset(&sascaga,',SASCAGAE);  /* Clear Map A. */

      /* Set the message attribute byte. */
   sascaga.msga = DFHBMASB;

      /* Move the message. */
   memcpy(&sascaga.msgo,&messages,strlen(messages));

      /* Send the map and return to CICS.*/
   EXEC CICS SEND MAP("SASCAGA") ERASE;
   EXEC CICS RETURN;

}              /* end of smnu() */
```

SASCBRW: Perform Browse Function

The SASCBRW program corresponds to the IBM programs DFH$*x*BRW (where *x* has the same values as mentioned earlier). It is the program that does the browse operation. It will start with the first record, or you can specify a starting place. The program can page forward and backward through the file.

```
#include <dfhbmsca.h> /* useful BMS definitions */

   /* C struct defining Map A, produced by DSECT2C */
#include <sascaga.h>

   /* C struct defining Map C, produced by DSECT2C */
#include <sascagc.h>
#include <ctype.h>    /* standard C type header */
#include <string.h>   /* standard C string handling header */

   /* general-purpose variables */
short i;
char rid[6] = "000000";  /* record key, the account number */
char ridb[6] = "000000"; /* record key, for backward paging */
char ridf[6] = "000000"; /* record key, for forward paging */

   /* indicates in which direction the browsing */
   /* is currently moving: F or B               */
char currop;

   /* indicates the previous browsing direction */
char lastop;
char status;             /* file status: 'H', 'L', or ' ' */
char messages[39];       /* message holding area */
short comlen;            /* length of CICS commarea */
char keynum[6];          /* record key, the account number */

   /* FILEA record layout */
struct FILEA{
   char stat;
   char numb[6];
   char name[20];
   char addrx[20];
   char phone[8];
   char datex[8];
   char amount[8];
   char comment[9];
} filea;

   /* function prototypes */
void errors(void);
void smnu(void);
void smsg(void);
void not_found(void);
void build_next(void);
void build_prev(void);
void page_forward(void);
void page_backward(void);
void receive(void);
void too_high(void);
void too_low(void);

   /* The main function begins here. */
main()
```

```
{

      /* Prepare to handle paging requests via PF keys. */
   EXEC CICS HANDLE AID CLEAR(smsg)
                        PF1(page_forward)
                        PF2(page_backward);

      /* Prepare to handle error conditions. */
   EXEC CICS HANDLE CONDITION ERROR(errors)
                              MAPFAIL(smsg)
                              NOTFND(not_found);

      /* Read the initial screen. */
   EXEC CICS RECEIVE MAP("SASCAGA");

      /* Was a starting account number provided? */
   if (memcmp(sascaga.keyl,"\0 \0",2) == 0){

         /* No, so default to zero. */
      strcpy(rid,"000000");
      strcpy(ridf,"000000");
   }
   else{
         /* Yes, so validate all of its digits. */
      for(i=0;i<=5;i++){
         if (isdigit((int) sascaga.KEYI[i]) == 0){
            strcpy(messages,"ACCOUNT NUMBER MUST BE NUMERIC");
            smnu();    /* There is no return from smnu(). */
         }
      }

         /* Save the starting account number for */
         /* use in either direction.             */
      memcpy(rid,sascaga.keyi,6);
      memcpy(ridf,sascaga.keyi,6);
      memcpy(ridb,sascaga.keyi,6);
   }

      /* Initiate the VSAM browse file operation. */
   EXEC CICS STARTBR DATASET("FILEA") RIDFLD(rid);

      /* Is the current account number 999999 */
      /* (the absolute maximum)?              */
   if (memcmp(rid,"999999",6) != 0){

         /* No, so page forward. */
      page_forward();
   }
   else{

         /* Yes, so set the status to HIGH and page backward. */
      status = 'H';
      page_backward();
   }
```

```
receive();    /* There is no return from receive(). */
}             /* end of main function */

   /* This function processes a forward paging request. */
void page_forward()
{
currop = 'F'; /* Set the indicator to forward paging mode. */

   /* Prepare to handle reading past the end of the file. */
EXEC CICS HANDLE CONDITION ENDFILE(too_high);

memset(&sascagc,'\0',SASCAGCE);   /* Clear Map C. */
memcpy(rid,ridf,6);
build_next();                     /* Read more records. */
memcpy(ridf,rid,6);

EXEC CICS SEND MAP("SASCAGC") ERASE; /* Display Map C. */
}                            /* end of page_forward function  */

   /* This function processes a backward paging request. */
void page_backward()
{
currop = 'B';   /* Set the indicator to backward paging mode. */

   /* Prepare to handle reading before the beginning of the file. */
EXEC CICS HANDLE CONDITION ENDFILE(too_low);

memset(&sascagc,',SASCAGCE);          /* Clear Map C. */
memcpy(rid,ridb,6);
memcpy(ridf,ridb,6);

   /* READPREV commands will reread the last record read by a    */
   /* READNEXT command, so perform an extra READPREV to account  */
   /* for this behavior.                                         */
if ((lastop == 'F') && (status != 'H')){
   EXEC CICS READPREV DATASET("FILEA") INTO(&filea) RIDFLD(rid)
             LENGTH(sizeof(struct FILEA)) KEYLENGTH(sizeof(rid));
}
build_prev();                        /* Read more records. */
memcpy(ridb,rid,6);

EXEC CICS SEND MAP("SASCAGC") ERASE; /* Display Map C. */
}                               /* end of page_backward function */

   /* This function controls the continual processing of     */
   /* page requests. Remember that the user can decide to    */
   /* press PF1 or PF2 at any time. Pressing these keys      */
   /* is an implicit request for forward or backward paging, */
   /* respectively. When a PF key is pressed, the            */
   /* appropriate function is driven; then control           */
   /* returns to the next executable statement within the    */
   /* for(;;) loop. From that point, normal FORWARD/BACKWARD */
   /* <ENTER> key processing continues.                      */
void receive()
```

```
{
   /* Continue to process page requests until the CLEAR key is   */
   /* pressed or the user does not enter a direction indicator.  */
for(;;){

      /* Remember which direction the reading is going. */
   lastop = currop;

      /* Read the latest screen. */
   EXEC CICS RECEIVE MAP("SASCAGC");

      /* When the MAPFAIL condition occurs here, the browse */
      /* operation ends, and  smsg() will be driven.        */
   status = ' ';

      /* Did the user enter a forward page request by keying an F ? */
   if (memcmp(sascagc.diri,"F",1) == 0){

         /* Yes, so cause forward paging to occur. */
      page_forward();
   }

      /* Did the user enter a backward page request by keying a B ? */
   else
      if (memcmp(sascagc.diri,"B",1) == 0){

            /* Yes, so cause backward paging to occur. */
         page_backward();
      }
      else{

            /* Neither F nor B was keyed, so resend the map. This */
            /* eventually results in a MAPFAIL condition and,     */
            /* therefore, the end of the browse operation.        */
         EXEC CICS SEND MAP("SASCAGC");
      }
   }                  /* end of for(;;) loop */
}                     /* end of receive() */


   /* This function is called when a read is attempted past */
   /* the end of file.                                      */
void too_high()
{
      /* Set status so that further reads are prevented. */
   status = 'H';
   memcpy(ridf,rid,6);
   memcpy(ridb,rid,6);

      /* Indicate that forward paging has ceased. */
   sascagc.diro = ' ';

      /* Prepare an informative message. */
   strcpy(sascagc.msg1o,"Hi-End of File");
```

```
   sascagc.msg1a= DFHBMASB;
}                     /* end of too_high() */

   /* This function is called when a read is attempted */
   /* before the beginning of the file.                */
void too_low()
{
      /* Set status so that further reads are prevented. */
   status = 'L';
   strcpy(ridf,"000000");
   strcpy(ridb,"000000");

      /* Indicate that backward paging has ceased. */
   sascagc.diro = ' ';

      /* Prepare an informative message. */
   strcpy(sascagc.msg2o,"Lo-End of File");
   sascagc.msg2a = DFHBMASB;
}                     /* end of too_low */

   /* This function is called when the specified starting       */
   /* account number is not found.  The VSAM browse             */
   /* operation is ended and appropriate messages are prepared. */
void not_found()
{
   strcpy(messages,"End of File - Please Restart");

   EXEC CICS ENDBR DATASET("FILEA");

   smnu();            /* There is no return from smnu() */
}                     /* end of not_found() */

   /* This function prepares an informative */
   /* message and calls smnu().             */
void smsg()
{
   strcpy(messages,"Press CLEAR to Exit");
   smnu();            /* There is no return from smnu(). */
}                     /* end of smsg() */

   /* This function is called when a severe and   */
   /* unrecoverable error is detected.  It        */
   /* causes a CICS transaction dump to be taken, */
   /* and prepares an error message.              */
void errors()
{
   EXEC CICS DUMP DUMPCODE("ERRS");

   strcpy(messages,"Transaction Terminated");
   smnu();            /* There is no return from smnu(). */
}                     /* end of errors() */
```

```
/* This function displays the main menu on */
   /* the screen and returns control to CICS. */
void smnu()
{    memset(&sascaga,'\0',SASCAGAE);    /* Clear Map A. */
     sascaga.msga = DFHBMASB;        /* Move the message to the map. */
     memcpy(sascaga.msgo,messages,strlen(messages));
     memset(messages,'\0',39);  /* Clear the message variable. */

     EXEC CICS SEND MAP("SASCAGA") ERASE;

     EXEC CICS RETURN;
 }
                          /* end of smnu() */

   /* This function performs up to four READNEXT            */
   /* commands to fill the screen in a forward paging mode. */
void build_next()
0
    int i;
        /* It takes four iterations to fill the screen. */
    for(i=1; i<= 4; i++){
        EXEC CICS READNEXT DATASET("FILEA") INTO(&filea) RIDFLD(rid)
                  LENGTH(sizeof(struct FILEA)) KEYLENGTH(sizeof(rid));
           /* If you read past the end of the file, STOP. */
        if (status == 'H')
           break;
        switch(i){
        case 1:{
             /* Move the record fields to line 1 of the map. */
           memcpy(sascagc.number1o,filea.numb,sizeof(filea.numb));
           memcpy(sascagc.name1o,filea.name,sizeof(filea.name));
           memcpy(sascagc.amount1o,filea.amount,sizeof(filea.amount));
           memcpy(ridb,rid,6);
           break;
        }
                        /* end of case: 1 */
        case 2:0
              /* Move the record fields to line 2 of the map. */
           memcpy(sascagc.number2o,filea.numb,sizeof(filea.numb));
           memcpy(sascagc.name2o,filea.name,sizeof(filea.name));
           memcpy(sascagc.amount2o,filea.amount,sizeof(filea.amount));
           break;
        }                  /* end of case: 2 */

        case 3:0
            /* Move the record fields to line 3 of the map. */
          memcpy(sascagc.number3o,filea.numb,sizeof(filea.numb));
          memcpy(sascagc.name3o,filea.name,sizeof(filea.name));
          memcpy(sascagc.amount3o,filea.amount,sizeof(filea.amount));
          break;
       }                 /* end of case: 3 */
       case 4:0
            /* Move the record fields to line 4 of the map. */
          memcpy(sascagc.number4o,filea.numb,sizeof(filea.numb));
```

```
          memcpy(sascagc.name4o,filea.name,sizeof(filea.name));
          memcpy(sascagc.amount4o,filea.amount,sizeof(filea.amount));
          break;
       }                /* end of case: 4 */
    }                   /* end of switch(i) *
    }                   /* end of for loop */ }
 }                      /* end of build_next() */


   /* This function performs up to four READPREV commands  */
   /* to fill the  screen in a backward paging mode.       */
void build_prev()
0
    int i;

        /* It takes 4 iterations to fill the screen. */
    for(i=1; i <= 4; i++)0
       EXEC CICS READPREV DATASET("FILEA") INTO(&filea) RIDFLD(rid)
                 LENGTH(sizeof(struct FILEA)) KEYLENGTH(sizeof(rid));

           /* If we read before the beginning of the file, STOP. */
       if (status == 'L')
           break;

           /* The records will be displayed in ascending order,  */
          /* so put the last one read at the top of the screen. */
       switch(i)0
          case 4:0
                /* Move the record fields to line 1 of the map. */
            memcpy(sascagc.number1o,filea.numb,sizeof(filea.numb));
            memcpy(sascagc.name1o,filea.name,sizeof(filea.name));
            memcpy(sascagc.amount1o,filea.amount,sizeof(filea.amount));
            break;
           }             /* end of case: 4 */
           case 3:0
                /* Move the record fields to line 2 of the map. */
            memcpy(sascagc.number2o,filea.numb,sizeof(filea.numb));
            memcpy(sascagc.name2o,filea.name,sizeof(filea.name));
            memcpy(sascagc.amount2o,filea.amount,sizeof(filea.amount));
            break;
          }              /* end of case: 3 */
           case 2:0
                /* Move the record fields to line 3 of the map. */
            memcpy(sascagc.number3o,filea.numb,sizeof(filea.numb));
            memcpy(sascagc.name3o,filea.name,sizeof(filea.name));
            memcpy(sascagc.amount3o,filea.amount,sizeof(filea.amount));
            break;
          }
              /* end of case: 2 */

          case 1:{
               /* Move the record fields to line 4 of the map. */
            memcpy(sascagc.number4o,filea.numb,sizeof(filea.numb));
            memcpy(sascagc.name4o,filea.name,sizeof(filea.name));
```

```
            memcpy(sascagc.amount4o,filea.amount,sizeof(filea.amount));
            break;

         }              /* end of case: 1 */
      }                 /* end of switch(i) */
   }                    /* end of for loop */
}                       /* end of build_prev() */
```

Index

A

abend handling 26, 70
access method parameters
  See amparms
all-resident C programs 6
all-resident libraries
  COOL option for 56
  SAS/C programs under CMS 53
  SAS/C programs under OS/390 TSO 39
all-resident load modules, creating 44
ALLRES parameter 44
ALLRESIDENT, COOL option 56
amparms 78
  file characteristics 78
  file usage 79
  for transient data 81
  JES spool files 83
ANSI Standard trigraphs, accepting 15
application program services, CICS 4
applications, C language
  See SAS/C programs
  See SAS/C programs under CMS
  See SAS/C programs under OS/390
applications, C++ language
  See C++ programs under CMS
  See C++ programs under OS/390
AUTO, COOL option 56
autocall libraries 59

B

Basic Mapping Support
  See BMS (Basic Mapping Support)
blksize amparm 78
  for JES spool files 84
  for transient data 82
BMS (Basic Mapping Support) 72
  DSECT2C utility 73
  example 75
  I/O operations 76
  LANG=C option 72
  maps and command translation 14
  mapsets 72
  SAS/C programs 23
  symbolic maps 72
BSD UNIX Socket Library 91

C

C++ input, specifying 57
C programs
  See SAS/C programs
  See SAS/C programs under CMS
  See SAS/C programs under OS/390
C++ programs under CMS 54
  compiling 54
  COOL options 55
  LCCCP EXEC 54
  LCXX EXEC 54
  linking 54
  preprocessing 54
C++ programs under OS/390 40
  all-resident load modules, creating 44
  compiling in batch mode 41, 47
  compiling with TSO 40
  COOL CLIST 41
  entry point, specifying 43
  LCCCL procedure 43
  LCCCP CLIST 40
  LCCCP procedure 41
  LCCCPC procedure 42
  LCCCPCL procedure 43
  LCCCXXL procedure 49
  LCCPCXX procedure 47
  LCCPCXXA procedure 50
  LCCPCXXL procedure 50
  LCXX CLIST 40
  linking in batch mode 41, 47
  linking with TSO 40
  preprocessing in batch mode 41, 47
  preprocessing with TSO 40
  program environment, specifying 43
  translating from C programs 41
CBMSMAPS option 14
character arguments 11
CICS, COOL option 57
CICS (Customer Information Control System) 2
  all-resident C programs 6
  application program services 4
  character arguments 11
  coding conventions 8
  command format 8
  components of 3
  control programs 4
  control tables 4
  cvda data type 10
  data-communication functions 3
  data-handling functions 4
  definition 1
  doubleword arguments 10
  fullword arguments 10
  halfword arguments 10
  hhmmss data type 10
  history of 2
  label data type 10
  monitoring functions 4
  multitasking 5
  multithreading 5
  name data type 10
  prototype generation 11
  re-entrancy 5
  releases supported by Command Language Translator 2
  SAS/C libraries 6
  SPE (Systems Programming Environment) 6
  system architecture 5
  system services 4
  transient library location 5
CICS file control 84
CICS option 14
cics.h header file 18
CICSVSE, COOL option 57
CLK370 CLIST 39
coding conventions 8
command format 8
Command Language Translator
  See SAS/C CICS Command Language Translator
command translation 11
  enabling 14
COMMAREA command 26
comments, nesting 14
_commptr variable 21
COMNEST option 14
compiling, transaction creation tutorial 32
compiling C++ programs
  CMS 54
  OS/390 batch 41, 47
  OS/390 TSO 40
compiling external CICS interface 63
compiling SAS/C programs
  CMS 52
  OS/390 batch 44, 45
  OS/390 TSO 38
control programs, CICS 4
control tables, CICS 4

COOL utility
  C++ programs under OS/390 TSO 41
  messages 60
  options 55
  SAS/C programs under CMS 52
  SAS/C programs under OS/390 TSO 39
cross-reference listings 15
cross-references, generating 58
Customer Information Control System
  See CICS (Customer Information Control System)
cvda data type 10
CXX, COOL option 57

D

data-communication functions, CICS 3
data declarations 21
data-handling functions, CICS 4
database support
  DL/I databases 85
  IMS/VS databases 85
  SQL databases 89
DEBUG option 14
debugging
  Command Language Translator 14
  DEBUG option 14
  SAS/C programs 27, 69
dfh2980.h header file 18
dfhaid.h header file 18
dfhbmsca.h header file 18
dfhcdblk.h header file 18
dfhmsrca.h header file 18
DFH$xALL program
  See SASCALL program
DFH$xBRW program
  See SASCBRW program
DFH$xMNU program
  See SASCMNU program
dib.h header file 18
_dibptr variable 21
DL/I database support 85
DLI option 14
dliuib.h header file 18
doubleword arguments 10
DPL Distributed Program Link 62
DSECT2C utility 73

E

EDF command interception 14
EDF option 14
eiblk.h header file 18
_eibptr variable 21
ENTRY, COOL option 57
ENTRY parameter 43
entry point, specifying 57
ENV parameter 43
environment variable support 26, 69
ENXREF, COOL option 58
error handling 24
EXEC CICS interface 63
EXEC DLI command processing 14
EXPAND option 14
extended name processing 58
external CICS interface 62
  CMS 64
  compiling 63
  EXEC CICS interface 63
  linking 63
  OS/390 batch 64
  OS/390 TSO 63
  translating 63
external references, resolving 56
EXTNAME, COOL option 58

F

file access method, specifying 78
file characteristic amparms 78
file usage amparms 79
filename specification 78
  JES spool files 83
  transient data 80
files
  See JES spool file I/O
  See header files
  See input files
  See output, file
FILES option 14
FLAG option 15
fullword arguments 10

G

GLOBAL, COOL option 58
global external variables 21
global TXTLIBs 57, 58

H

halfword arguments 10
HANDLE AID command 24
HANDLE CONDITION command 24
header files 18
_heap option 68
hhmmss data type 10

I

I/O functions
  JES spool files 82
  SAS/C 26
  TCP/IP Socket Library 96
IMS/VS database support 85
indep compiler option 18
input files 37
  JES spool files 82
  SYSIN files 37
interval control 26

J

JAPAN option 15
JES spool file I/O 82
  amparms 83
  blksize amparm 84
  CICS file control 84
  filename specification 83
  input files 82
  ISSUE commands, record ID 84
  output files 82
  recfm amparm 83
  reclen amparm 83

L

label data type 10
LANG=C option 72
LC370 CLIST 38
LC370 EXEC 52
LCCCL procedure 43, 45
LCCCP CLIST 37, 40
LCCCP EXEC 51, 54
LCCCP procedure 41
LCCCPC procedure 42
LCCCPCL procedure 43, 44
LCCCXXL procedure 49
LCCPCXX procedure 47
LCCPCXXA procedure 50
LCCPCXXL procedure 50
LCXX CLIST 40
LCXX EXEC 54
LIB, COOL option 59
LINK command 26
link-editing, transaction creation tutorial 32
linkage editor, invoking 59
_linkage option 68
linking all-resident libraries
  COOL option for 56
  SAS/C programs under CMS 53
  SAS/C programs under OS/390 TSO 39
linking C++ programs
  CMS 54
  OS/390 batch 41, 47
  OS/390 TSO 40
linking external CICS interface 63
linking SAS/C programs
  CMS, for VSE 61
  CMS, with CMS EXECs 53
  OS/390, for VSE 62
  OS/390 batch 44
  OS/390 TSO 39
  OS/390 TSO, all-resident programs 39
  OS/390 TSO, CLK370 CLIST 39
  OS/390 TSO, with TSO CLISTs 39
linking with CMS EXECs 53
listings
  See output, printed
LKED, COOL option 59
LOAD, COOL option 59
LOAD command 26
LOADLIB, COOL option 59

M

main() function, passing arguments to 23
$MAIN0 entry point 69

mapsets 72
message levels, setting 15
messages
  COOL 60
  diagnostic 66
  directing to a terminal 15, 60
  enabling 60
  uppercasing 60
_mneed option 68
monitoring functions, CICS 4
multitasking 5
multithreading 5

N

name data type 10
_negopts option 68

O

_options option 68
OPTIONS option 15
OUTLRECL option 15
output, file 82
  JES spool files 82
  load module name, specifying 59
  record format 15
  record length 15
  sequence numbers 15
  SYSPUNCH files 37
  SYSTERM files 37
  translator output files 51
output, printed 15
  cross-reference listings 15
  destination, specifying 60
  formatted source listings 15
  generating 15
  lines per page 15
  SAS/C programs under CMS 51
  SYSPRINT files 37
  uppercasing text 15
output, terminal 15, 51, 60
OUTRECFM option 15
OUTSEQ option 15
OVERSTRIKE option 15

P

page amparm 79
PAGESIZE option 15
#pragma options statement 13
PREM, COOL option 60
preprocessing, transaction creation tutorial 32
preprocessing C++ programs
  CMS 54
  OS/390 batch 41, 47
  OS/390 TSO 40
preprocessing SAS/C programs
  OS/390 batch 44
  OS/390 TSO 37, 39
PRINT, COOL option 60
print amparm 79
PRINT option 15
printed output
  See output, printed
program control 26
program environment, specifying 43
programs, C language
  See SAS/C programs
  See SAS/C programs under CMS
  See SAS/C programs under OS/390
programs, C++ language
  See C++ programs under CMS
  See C++ programs under OS/390
PROTO option 15
prototype generation 11
  enabling 15
  uses for 13
pseudoregisters, removing 60

R

re-entrancy 5
recfm amparm 78
  for JES spool files 83
  for transient data 81
reclen amparm 78
  for JES spool files 83
  for transient data 81
record format, translator output files 15
record length, translator output files 15
RELEASE command 26
reports
  See output, printed
RETURN command 26
run-time options 68

S

SAS/C CICS Command Language Translator 2
  ANSI Standard trigraphs, accepting 15
  BMS maps and command translation 14
  C code, displaying 14
  C code, uppercasing 15
  CBMSMAPS option 14
  character arguments 11
  CICS option 14
  CICS releases supported 2
  coding conventions 8
  command format 8
  command translation 11
  command translation, enabling 14
  comments, nesting 14
  COMNEST option 14
  cvda data type 10
  DEBUG option 14
  debugging 14
  DLI option 14
  doubleword arguments 10
  EDF command interception 14
  EDF option 14
  EXEC DLI command processing 14
  EXPAND option 14
  FILES option 14
  FLAG option 15
  fullword arguments 10
  halfword arguments 10
  hhmmss data type 10
  JAPAN option 15
  label data type 10
  message levels, setting 15
  messages, directing to a terminal 15
  name data type 10
  OPTIONS option 15
  OUTLRECL option 15
  OUTRECFM option 15
  OUTSEQ option 15
  OVERSTRIKE option 15
  PAGESIZE option 15
  #pragma options statement 13
  PRINT option 15
  PROTO option 15
  SOURCE option 15
  special characters, printing as overstrikes 15
  special characters, translating 15
  SYS file prefix, replacing 14
  TERM option 15
  TRANS option 15
  translator options, descriptions 14
  translator options, listing 15
  translator options, specifying 13
  TRIGRAPHS option 15
  UPPER option 15
  XREF option 15
SAS/C libraries, and CICS 6
SAS/C library functions 18
SAS/C programs 17
  abend handling 26, 70
  BMS definitions 23
  data declarations 21
  debugging 27
  displaying code 14
  efficiency 69
  environment variable support 26, 69
  error handling 24
  global external variables 21
  HANDLE AID command 24
  HANDLE CONDITION command 24
  header files 18
  I/O functions 26
  indep compiler option 18
  interval control 26
  main() function, passing arguments to 23
  $MAIN0 entry point 69
  preparing for CICS 67
  program control 26
  run-time options 68
  SAS/C library functions 18
  SQL statements 64
  task control 26
  translating to C++ 41
  uppercasing code 15
SAS/C programs under CMS 50
  compiling 52
  COOL utility 52
  LC370 EXEC 52
  LCCCP EXEC 51
  linkage editor control statements 52
  linking all-resident programs 53
  linking for VSE 61
  linking with CMS EXECs 53
  listing files 51
  translator input files 51

SAS/C programs under OS/390 36
  all-resident programs 39
  CLK370 CLIST 39
  compiling in batch mode 44, 45
  compiling with TSO 38
  LC370 CLIST 38
  LCCCL procedure 45
  LCCCP CLIST 37
  LCCCPCL procedure 44
  linking for VSE 62
  linking in batch mode 44
  linking with COOL 39
  linking with TSO 39
  linking with TSO CLISTs 39
  linking without COOL 39
  preprocessing in batch mode 44
  preprocessing with TSO 37, 39
  TSO CLISTs 39
SAS/C Socket Library 92
SASCALL program 100
SASCBRW program 109
SASCMNU program 100
screen definition 72
  See also BMS (Basic Mapping Support)
  browse function 109
  inquiry functions 100
  main menu 100
  SASCALL program 100
  SASCBRW program 109
  SASCMNU program 100
  update functions 100
sequence numbers, translator output files 15
socket libraries
  BSD UNIX Socket Library 91
  SAS/C Socket Library 92
  TCP/IP Socket Library 92
source listings
  See output, printed
SOURCE option 15
SPE, COOL option 60
SPE libraries 60
SPE (Systems Programming Environment) 6
special characters
  printing as overstrikes 15
  translating 15
SQL database support 89
SQL statements in SAS/C programs 64
_stack option 68
symbolic maps 72
SYS file prefix, replacing 14
SYSIN files 37
SYSPRINT files 37
SYSPUNCH files 37
system services, CICS 4
Systems Programming Environment (SPE) 6
SYSTERM files 37

T

takesocket function 94
task control 26
TCP/IP Socket Library
  communication functions 93
  I/O functions 96
  resident functions 92
  takesocket function 94
  unsupported functions 96
TCP/IP (Transport Control Protocol/Internet Protocol) 91
TERM, COOL option 60
TERM option 15
terminal control
  See BMS (Basic Mapping Support)
terminals, directing messages to
  See output, terminal
TRANS option 15
transaction creation tutorial 29
  CICS and the example code 33
  compiling source code 32
  example code 30
  link-editing source code 32
  preprocessing source code 32
transient data 80
  amparms 81
  filename specification 80
  positioning queues 80
transient library location 5
translator options
  descriptions 14
  listing 15
  specifying 13
Transport Control Protocol/Internet Protocol (TCP/IP) 91
TRIGRAPHS option 15

U

UPPER, COOL option 60
UPPER option 15
uppercasing text 15

V

VSE
  CICSVSE, COOL option 57
  linking SAS/C programs under CMS 61
  linking SAS/C programs under OS/390 62

W

WARN, COOL option 60

X

XCTL command 26
XREF option 15

Your Turn

If you have comments or suggestions about SAS/C® CICS User's Guide, Release 7.00, please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
email: yourturn@sas.com

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
email: suggest@sas.com

*Welcome * Bienvenue * Willkommen * Yohkoso * Bienvenido*

SAS Publishing Is Easy to Reach

Visit our Web page located at www.sas.com/pubs

You will find product and service details, including

- **sample chapters**
- **tables of contents**
- **author biographies**
- **book reviews**

Learn about

- **regional user-group conferences**
- **trade-show sites and dates**
- **authoring opportunities**
- **custom textbooks**

Explore all the services that SAS Publishing has to offer!

Your Listserv Subscription Automatically Brings the News to You

Do you want to be among the first to learn about the latest books and services available from SAS Publishing? Subscribe to our listserv **newdocnews-l** and, once each month, you will automatically receive a description of the newest books and which environments or operating systems and SAS® release(s) each book addresses.

To subscribe,

1. Send an e-mail message to **listserv@vm.sas.com**.
2. Leave the "Subject" line blank.
3. Use the following text for your message:

   subscribe NEWDOCNEWS-L ***your-first-name your-last-name***

   For example: subscribe NEWDOCNEWS-L John Doe

Create Customized Textbooks Quickly, Easily, and Affordably

SelecText® offers instructors at U.S. colleges and universities a way to create custom textbooks for courses that teach students how to use SAS software.

For more information, see our Web page at **www.sas.com/selectext**, or contact our SelecText coordinators by sending e-mail to **selectext@sas.com**.

You're Invited to Publish with SAS Institute's User Publishing Program

If you enjoy writing about SAS software and how to use it, the User Publishing Program at SAS Institute offers a variety of publishing options. We are actively recruiting authors to publish books, articles, and sample code. Do you find the idea of writing a book or an article by yourself a little intimidating? Consider writing with a co-author. Keep in mind that you will receive complete editorial and publishing support, access to our users, technical advice and assistance, and competitive royalties. Please contact us for an author packet. E-mail us at **sasbbu@sas.com** or call 919-531-7447. See the SAS Publishing Web page at **www.sas.com/pubs** for complete information.

Book Discount Offered at SAS Public Training Courses!

When you attend one of our SAS Public Training Courses at any of our regional Training Centers in the U.S., you will receive a 20% discount on book orders that you place during the course. Take advantage of this offer at the next course you attend!

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
Fax 919-677-4444

E-mail: sasbook@sas.com
Web page: www.sas.com/pubs
To order books, call Fulfillment Services at 800-727-3228*
For other SAS business, call 919-677-8000*

*** Note:** Customers outside the U.S. should contact their local SAS office.